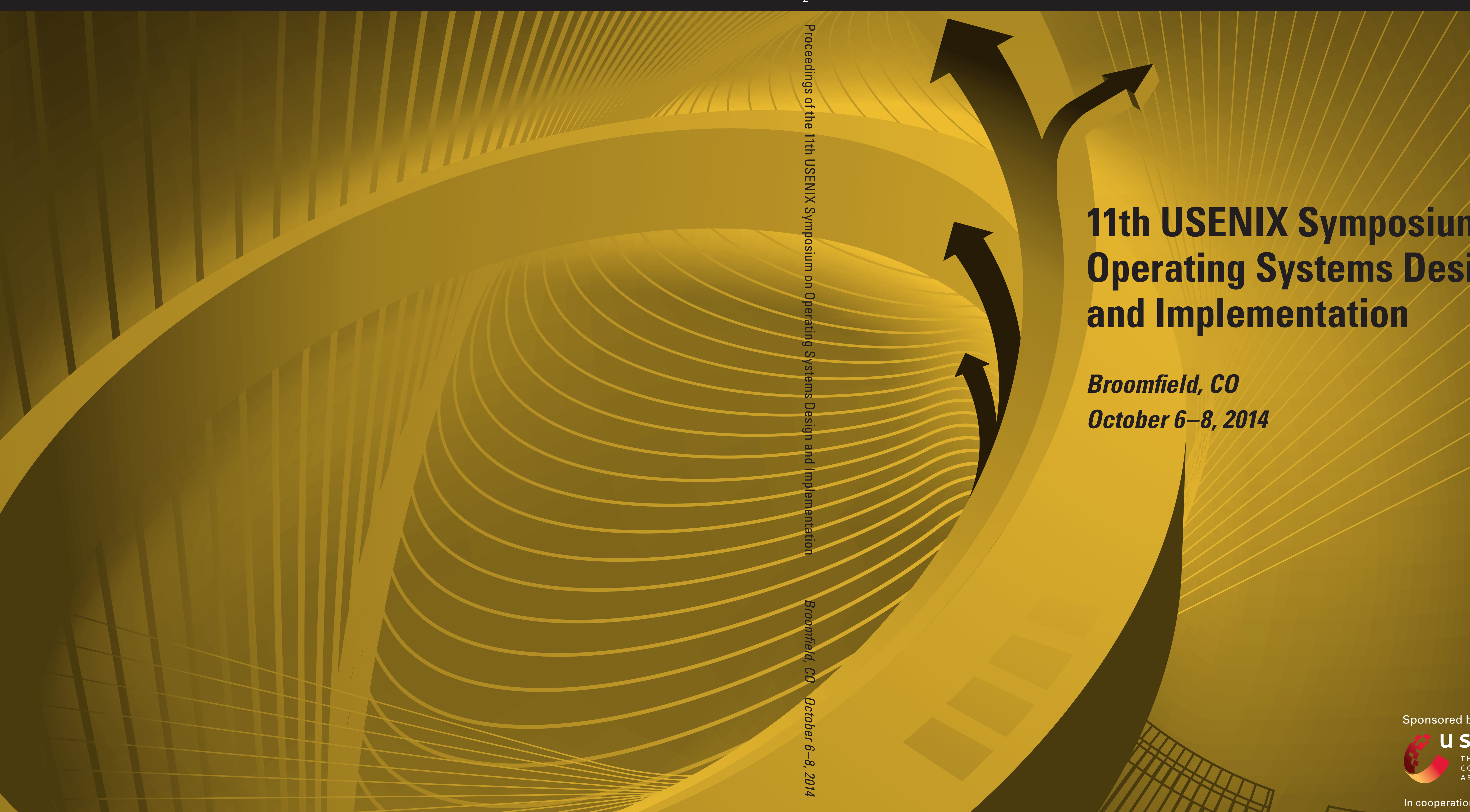
conference proceedings



11th USENIX Symposium on Operating Systems Design and Implementation

Broomfield, CO

October 6–8, 2014

Sponsored by usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with ACM SIGOPS

Thanks to Our OSDI '14 Sponsors

Diamond Sponsors

Gold Sponsors

Silver Sponsors

Bronze Sponsors

General Sponsor

Media Sponsors and Industry Partners

ACM *Queue*
ADMIN magazine
CRC Press
Distributed Management Task Force (DMTF)
The FreeBSD Foundation
HPCwire
InfoSec News
Linux Pro Magazine
LXer
No Starch Press
O'Reilly Media
Raspberry Pi Geek
UserFriendly.org

© 2014 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-931971-16-4

Thanks to Our USENIX and LISA SIG Supporters

USENIX Patrons

Google Microsoft Research NetApp VMware

USENIX Benefactors

Akamai Facebook Hewlett-Packard IBM Research
Linux Pro Magazine Puppet Labs

USENIX and LISA SIG Partners

Cambridge Computer Google

USENIX Partners

Can Stock Photo EMC

USENIX Association

Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)

October 6–8, 2014
Broomfield, CO

Conference Organizers

Program Co-Chairs
Jason Flinn, *University of Michigan*
Hank Levy, *University of Washington*

Program Committee
Lorenzo Alvisi, *The University of Texas at Austin*
Dave Andersen, *Carnegie Mellon University*
Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
Mihai Budiu, *Microsoft Research*
George Candea, *École Polytechnique Fédérale de Lausanne (EPFL)*
Peter Chen, *University of Michigan*
Allen Clement, *Google and Max Planck Institute for Software Systems (MPI-SWS)*
Landon Cox, *Duke University*
Nick Feamster, *Georgia Institute of Technology*
Bryan Ford, *Yale University*
Roxana Geambasu, *Columbia University*
Gernot Heiser, *University of New South Wales Australia/NICTA*
M. Frans Kaashoek, *MIT CSAIL*
Ed Nightingale, *Microsoft Research*
Timothy Roscoe, *ETH Zürich*
Emin Gün Sirer, *Cornell University*
Doug Terry, *Microsoft Research*
Geoff Voelker, *University of California, San Diego*
Andrew Warfield, *University of British Columbia*
Emmett Witchel, *The University of Texas at Austin*
Junfeng Yang, *Columbia University*
Yuanyuan Zhou, *University of California, San Diego*
Willy Zwaenepoel, *École Polytechnique Fédérale de Lausanne (EPFL)*

Poster Session Co-Chairs
Allen Clement, *Google and Max Planck Institute for Software Systems (MPI-SWS)*
Roxana Geambasu, *Columbia University*

Steering Committee
Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
Brad Chen, *Google*
Casey Henderson, *USENIX*
Brian Noble, *University of Michigan*
Margo Seltzer, *Harvard School of Engineering and Applied Sciences and Oracle*
Chandu Thekkath, *Microsoft Research Silicon Valley*
Amin Vahdat, *Google and University of California, San Diego*

External Review Committee
Atul Adya, *Google*
Emery Berger, *University of Massachusetts*
Luis Ceze, *University of Washington*
Angela Demke Brown, *University of Toronto*
Greg Ganger, *Carnegie Mellon University*
Joseph Gonzalez, *University of California, Berkeley*
Andreas Haeberlen, *University of Pennsylvania*
Galen Hunt, *Microsoft Research*
Sam King, *Twitter*
Eddie Kohler, *Harvard University*
Ramakrishna Kotla, *Microsoft Research*
Jinyang Li, *New York University*
Wyatt Lloyd, *Facebook and University of Southern California*
Shan Lu, *University of Wisconsin—Madison*
Jeff Mogul, *Google*
Satish Narayanasamy, *University of Michigan*
Jason Nieh, *Columbia University*
Vivek Pai, *Princeton University*
Rodrigo Rodrigues, *CITI/NOVA University of Lisbon*
Bianca Schroeder, *University of Toronto*
Mike Swift, *University of Wisconsin—Madison*
Kaushik Veeraraghavan, *Facebook*
Hakim Weatherspoon, *Cornell University*
Matt Welsh, *Google*
John Wilkes, *Google*
Ding Yuan, *University of Toronto*
Nickolai Zeldovich, *MIT CSAIL*
Feng Zhao, *Microsoft Research*

External Reviewers

Radu Banabic
Julian Bangert
Pramod Bhatotia
Stefan Bucur
Haogang Chen
Vitaly Chipounov
Austin Clements
David Cock
Patrick Colp
Cody Cutler
Ricardo Dias
Pedro Fonseca
João Garcia
Qian Ge
Zhenyu Guo
Ji Hong
Anuj Kalia
Manos Kapritsos
Baris Kasikci
Volodymyr Kuznetsov
David Lazar
Geoffrey Lefebvre
João Leitão
Hyeontaek Lim
Yunxin Liu
Yandong Mao
Syed Akbar Mehdi
Dutch Meyer
Henrique Moniz
Iulian Moraru
Toby Murray
Mihir Nanavati
Neha Narula
Daniel Peek
Raluca Ada Popa
Daniel Porto
Dan Ports
Zhengping Qian
Shriram Rajagopalan
Franzi Roesner
Chunzhi Su
Philippe Suter
Stephen Tu
Jelle van den Hooff
Jonas Wagner
Xi Wang
Yang Wang
Ming Wu
Chao Xie
Fan Yang
Cristian Zamfir
Huanchen Zhang
Timmy Zhu

11th USENIX Symposium on Operating Systems Design and Implementation
October 6–8, 2014
Broomfield, CO

Message from the Program Chair . . . . **ix**

Monday, October 6, 2014

Who Put the Kernel in My OS Conference?

Arrakis: The Operating System is the Control Plane **1**
Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, and Thomas Anderson, *University of Washington;* Timothy Roscoe, *ETH Zürich*

Decoupling Cores, Kernels, and Operating Systems **17**
Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe, *ETH Zürich*

Jitk: A Trustworthy In-Kernel Interpreter Infrastructure **33**
Xi Wang, David Lazar, Nickolai Zeldovich, and Adam Chlipala, *MIT CSAIL;* Zachary Tatlock, *University of Washington*

IX: A Protected Dataplane Operating System for High Throughput and Low Latency **49**
Adam Belay, *Stanford University;* George Prekas, *École Polytechnique Fédérale de Lausanne (EPFL);* Ana Klimovic, Samuel Grossman, and Christos Kozyrakis, *Stanford University;* Edouard Bugnion, *École Polytechnique Fédérale de Lausanne (EPFL)*

Data in the Abstract

Willow: A User-Programmable SSD **67**
Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, *University of California, San Diego*

Physical Disentanglement in a Container-Based File System **81**
Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *University of Wisconsin—Madison*

Customizable and Extensible Deployment for Mobile/Cloud Applications **97**
Irene Zhang, Adriana Szekeres, Dana Van Aken, and Isaac Ackerman, *University of Washington;* Steven D. Gribble, *Google and University of Washington;* Arvind Krishnamurthy and Henry M. Levy, *University of Washington*

Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems **113**
Riley Spahn and Jonathan Bell, *Columbia University;* Michael Lee, *The University of Texas at Austin;* Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser, *Columbia University*

My Insecurities

Protecting Users by Confining JavaScript with COWL **131**
Deian Stefan and Edward Z. Yang, *Stanford University;* Petr Marchenko, *Google;* Alejandro Russo, *Chalmers University of Technology;* Dave Herman, *Mozilla;* Brad Karp, *University College London;* David Mazières, *Stanford University*

Code-Pointer Integrity **147**
Volodymyr Kuznetsov, *École Polytechnique Fédérale de Lausanne (EPFL);* László Szekeres, *Stony Brook University;* Mathias Payer, *Purdue University;* George Candea, *École Polytechnique Fédérale de Lausanne (EPFL);* R. Sekar, *Stony Brook University;* Dawn Song, *University of California, Berkeley*

(Monday, October 6, continues on next page)

Ironclad Apps: End-to-End Security via Automated Full-System Verification 165
Chris Hawblitzel, Jon Howell, and Jacob R. Lorch, *Microsoft Research;* Arjun Narayan, *University of Pennsylvania;* Bryan Parno, *Microsoft Research;* Danfeng Zhang, *Cornell University;* Brian Zill, *Microsoft Research*

Shill: A Secure Shell Scripting Language 183
Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong, *Harvard University*

Variety Pack

GPUnet: Networking Abstractions for GPU Programs 201
Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, and Emmett Witchel, *The University of Texas at Austin;* Amir Wated and Mark Silberstein, *Technion—Israel Institute of Technology*

***The Mystery Machine:* End-to-end Performance Analysis of Large-scale Internet Services** 217
Michael Chow, *University of Michigan;* David Meisner, *Facebook, Inc.;* Jason Flinn, *University of Michigan;* Daniel Peek, *Facebook, Inc.;* Thomas F. Wenisch, *University of Michigan*

End-to-end Performance Isolation Through Virtual Datacenters 233
Sebastian Angel, *The University of Texas at Austin;* Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska, *Microsoft Research*

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems 249
Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, *University of Toronto*

Tuesday, October 7, 2014

Head in the Cloud

Shielding Applications from an Untrusted Cloud with Haven 267
Andrew Baumann, Marcus Peinado, and Galen Hunt, *Microsoft Research*

Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing 285
Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, and Jingren Zhou, *Microsoft;* Zhengping Qian, Ming Wu, and Lidong Zhou, *Microsoft Research*

The Power of Choice in Data-Aware Cluster Scheduling 301
Shivaram Venkataraman and Aurojit Panda, *University of California, Berkeley;* Ganesh Ananthanarayanan, *Microsoft Research;* Michael J. Franklin and Ion Stoica, *University of California, Berkeley*

Heading Off Correlated Failures through Independence-as-a-Service 317
Ennan Zhai, *Yale University;* Ruichuan Chen, *Bell Labs and Alcatel-Lucent;* David Isaac Wolinsky and Bryan Ford, *Yale University*

Storage Runs Hot and Cold

Characterizing Storage Workloads with Counter Stacks 335
Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield, *Coho Data*

Pelican: A Building Block for Exascale Cold Data Storage 351
Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, and Sergey Legtchenko, *Microsoft Research;* Aaron Ogus, *Microsoft;* Eric Peterson and Antony Rowstron, *Microsoft Research*

A Self-Configurable Geo-Replicated Cloud Storage System 367
Masoud Saeida Ardekani, *INRIA and Sorbonne Universités;* Douglas B. Terry, *Microsoft Research*

f4: Facebook's Warm BLOB Storage System 383
Subramanian Muralidhar, *Facebook, Inc.;* Wyatt Lloyd, *University of Southern California and Facebook, Inc.;* Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, and Viswanath Sivakumar, *Facebook, Inc.;* Linpeng Tang, *Princeton University and Facebook, Inc.*; Sanjeev Kumar, *Facebook, Inc.*

Pest Control

SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems 399
Tanakorn Leesatapornwongsa and Mingzhe Hao, *University of Chicago;* Pallavi Joshi, *NEC Labs America;* Jeffrey F. Lukman, *Surya University;* Haryadi S. Gunawi, *University of Chicago*

Ski: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration 415
Pedro Fonseca, *Max Planck Institute for Software Systems (MPI-SWS);* Rodrigo Rodrigues, *CITI/NOVA University of Lisbon;* Björn B. Brandenburg, *Max Planck Institute for Software Systems (MPI-SWS)*

All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications 433
Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *University of Wisconsin–Madison*

Torturing Databases for Fun and Profit 449
Mai Zheng, *The Ohio State University;* Joseph Tucek, *HP Labs;* Dachuan Huang and Feng Qin, *The Ohio State University;* Mark Lillibridge, Elizabeth S. Yang, and Bill W. Zhao, *HP Labs;* Shashank Singh, *The Ohio State University*

Transaction Action

Fast Databases with Fast Durability and Recovery Through Multicore Parallelism 465
Wenting Zheng and Stephen Tu, *Massachusetts Institute of Technology;* Eddie Kohler, *Harvard University;* Barbara Liskov, *Massachusetts Institute of Technology*

Extracting More Concurrency from Distributed Transactions 479
Shuai Mu, *Tsinghua University and New York University;* Yang Cui and Yang Zhang, *New York University;* Wyatt Lloyd, *University of Southern California and Facebook, Inc.;* Jinyang Li, *New York University*

Salt: Combining ACID and BASE in a Distributed Database 495
Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan, *The University of Texas at Austin*

Phase Reconciliation for Contended In-Memory Transactions 511
Neha Narula and Cody Cutler, *MIT CSAIL;* Eddie Kohler, *Harvard University;* Robert Morris, *MIT CSAIL*

Wednesday, October 8, 2014

Play It Again, Sam

Eidetic Systems 525
David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen, *University of Michigan*

Detecting Covert Timing Channels with Time-Deterministic Replay 541
Ang Chen, *University of Pennsylvania;* W. Brad Moore, *Georgetown University;* Hanjun Xiao, Andreas Haeberlen, and Linh Thi Xuan Phan, *University of Pennsylvania;* Micah Sherr and Wenchao Zhou, *Georgetown University*

Identifying Information Disclosure in Web Applications with Retroactive Auditing 555
Haogang Chen, Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek, *MIT CSAIL*

(Wednesday, October 8, continues on next page)

Help Me Learn

Project Adam: Building an Efficient and Scalable Deep Learning Training System 571
Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman, *Microsoft Research*

Scaling Distributed Machine Learning with the Parameter Server 583
Mu Li, *Carnegie Mellon University and Baidu;* David G. Andersen and Jun Woo Park, *Carnegie Mellon University;* Alexander J. Smola, *Carnegie Mellon University and Google, Inc.;* Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su, *Google, Inc.*

GraphX: Graph Processing in a Distributed Dataflow Framework 599
Joseph E. Gonzalez, *University of California, Berkeley;* Reynold S. Xin, *University of California, Berkeley, and Databricks;* Ankur Dave, Daniel Crankshaw, and Michael J. Franklin, *University of California, Berkeley;* Ion Stoica, *University of California, Berkeley, and Databricks*

Hammers and Saws

Nail: A Practical Tool for Parsing and Generating Data Formats 615
Julian Bangert and Nickolai Zeldovich, *MIT CSAIL*

***lprof:* A Non-intrusive Request Flow Profiler for Distributed Systems** 629
Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm, *University of Toronto*

Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud 645
Stefan C. Müller, *ETH Zürich and University of Applied Sciences Northwestern Switzerland;* Gustavo Alonso and Adam Amara, *ETH Zürich;* André Csillaghy, *University of Applied Sciences Northwestern Switzerland*

User-Guided Device Driver Synthesis 661
Leonid Ryzhyk, *University of Toronto, NICTA, and University of New South Wales;* Adam Walker, *NICTA and University of New South Wales;* John Keys, *Intel Corporation;* Alexander Legg, *NICTA and University of New South Wales;* Arun Raghunath, *Intel Corporation;* Michael Stumm, *University of Toronto;* Mona Vij, *Intel Corporation*

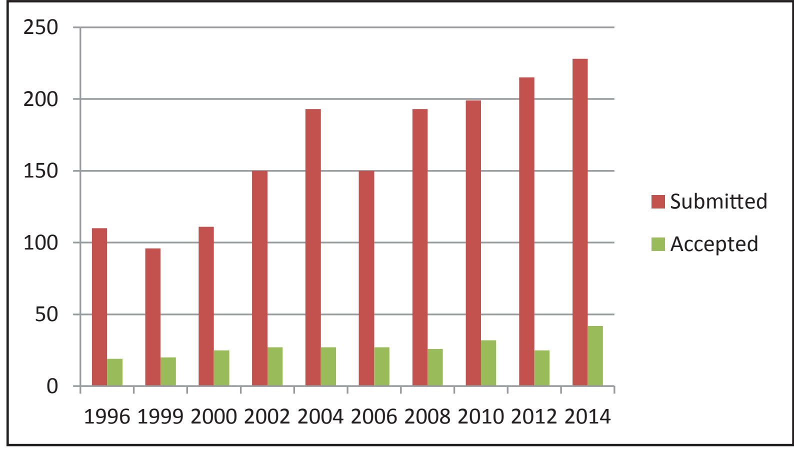
Message from the
OSDI ’14 Program Co-Chairs

We are delighted to welcome you to the 11th USENIX Symposium on Operating Systems Design and Implementation. This year’s program contains 42 papers representing some of the best research from academia and industry in the area of experimental systems.

This year, we received 228 submissions that met the formatting guidelines for the conference. We used three rounds of review, followed by electronic discussion after each round, to evaluate the submissions. Eleven papers were designated as “quick-accepts” based on reviews and electronic discussions. An additional 66 papers were discussed during a day-and-a-half-long PC meeting of which 31 were selected for the program.

For the first time at OSDI, the submission process included a response period in which authors could answer reviewer questions and address factual errors in the reviews. Responses had a measurable impact on PC meeting discussions, helping some papers and hurting others. Overall, we believe responses were quite useful in improving the fairness of the review process and the quality of the selected program.

From the beginning of the process, our goal was to increase the number of papers appearing at OSDI to approximately 40 accepted papers. As the chart below shows, the number of submissions to the conference has steadily increased over the years, while the number of accepted papers has not kept pace. At the same time, flagship conferences in related fields (architecture, networking, programming languages, etc.) have all substantially increased the number of accepted papers. This harms our field in several ways: some of our top work does not appear at our flagship conferences, our researchers are at a competitive disadvantage compared to researchers in other fields, and the review process becomes excessively negative. We believe that the 42 papers in the final program reflect both the overall strength and the breadth of research in our community.



We wish to thank the members of the program committee and the external review committee who produced 950 reviews in approximately 2 months time. We especially thank the many members of the external review committee who volunteered to do extra reviews when the number of submissions came in higher than we expected. We also thank the external reviewers who helped with reviews and the USENIX staff who organized the conference.

Jason Flinn, ***University of Michigan***
Hank Levy, ***University of Washington***
OSDI ’14 Program Co-Chairs

Arrakis: The Operating System is the Control Plane

Simon Peter* Jialin Li* Irene Zhang* Dan R. K. Ports* Doug Woos*
Arvind Krishnamurthy* Thomas Anderson* Timothy Roscoe†
*University of Washington** *ETH Zurich*†

Abstract

Recent device hardware trends enable a new approach to the design of network server operating systems. In a traditional operating system, the kernel mediates access to device hardware by server applications, to enforce process isolation as well as network and disk security. We have designed and implemented a new operating system, Arrakis, that splits the traditional role of the kernel in two. Applications have direct access to virtualized I/O devices, allowing most I/O operations to skip the kernel entirely, while the kernel is re-engineered to provide network and disk protection without kernel mediation of every operation. We describe the hardware and software changes needed to take advantage of this new abstraction, and we illustrate its power by showing improvements of 2-5× in latency and 9× in throughput for a popular persistent NoSQL store relative to a well-tuned Linux implementation.

1 Introduction

Reducing the overhead of the operating system process abstraction has been a longstanding goal of systems design. This issue has become particularly salient with modern client-server computing. The combination of high speed Ethernet and low latency persistent memories is considerably raising the efficiency bar for I/O intensive software. Many servers spend much of their time executing operating system code: delivering interrupts, demultiplexing and copying network packets, and maintaining file system meta-data. Server applications often perform very simple functions, such as key-value table lookup and storage, yet traverse the OS kernel multiple times per client request.

These trends have led to a long line of research aimed at optimizing kernel code paths for various use cases: eliminating redundant copies in the kernel [45], reducing the overhead for large numbers of connections [27], protocol specialization [44], resource containers [8, 39], direct transfers between disk and network buffers [45], interrupt steering [46], system call batching [49], hardware TCP acceleration, etc. Much of this has been adopted in mainline commercial OSes, and yet it has been a losing battle: we show that the Linux network and file system stacks have latency and throughput many times worse than that achieved by the raw hardware.

Twenty years ago, researchers proposed streamlining packet handling for parallel computing over a network of workstations by mapping the network hardware directly into user space [19, 22, 54]. Although commercially unsuccessful at the time, the virtualization market has now led hardware vendors to revive the idea [6, 38, 48], and also extend it to disks [52, 53].

This paper explores the OS implications of removing the kernel from the data path for nearly all I/O operations. We argue that doing this must provide applications with the same security model as traditional designs; it is easy to get good performance by extending the trusted computing base to include application code, e.g., by allowing applications unfiltered direct access to the network/disk.

We demonstrate that operating system protection is not contradictory with high performance. For our prototype implementation, a client request to the Redis persistent NoSQL store has 2× better read latency, 5× better write latency, and 9× better write throughput compared to Linux.

We make three specific contributions:

- We give an architecture for the division of labor between the device hardware, kernel, and runtime for direct network and disk I/O by unprivileged processes, and we show how to efficiently emulate our model for I/O devices that do not fully support virtualization (§3).
- We implement a prototype of our model as a set of modifications to the open source Barrelfish operating system, running on commercially available multi-core computers and I/O device hardware (§3.8).
- We use our prototype to quantify the potential benefits of user-level I/O for several widely used network services, including a distributed object cache, Redis, an IP-layer middlebox, and an HTTP load balancer (§4). We show that significant gains are possible in terms of both latency and scalability, relative to Linux, in many cases without modifying the application programming interface; additional gains are possible by changing the POSIX API (§4.3).

2 Background

We first give a detailed breakdown of the OS and application overheads in network and storage operations today, followed by a discussion of current hardware technologies that support user-level networking and I/O virtualization.

To analyze the sources of overhead, we record timestamps at various stages of kernel and user-space processing. Our experiments are conducted on a six machine cluster consisting of 6-core Intel Xeon E5-2430 (Sandy Bridge) systems at 2.2 GHz running Ubuntu Linux 13.04.

		Linux				Arrakis			
		Receiver running		CPU idle		Arrakis/P		Arrakis/N	
Network stack	in	1.26	(37.6%)	1.24	(20.0%)	0.32	(22.3%)	0.21	(55.3%)
	out	1.05	(31.3%)	1.42	(22.9%)	0.27	(18.7%)	0.17	(44.7%)
Scheduler		0.17	(5.0%)	2.40	(38.8%)	-		-	
Copy	in	0.24	(7.1%)	0.25	(4.0%)	0.27	(18.7%)	-	
	out	0.44	(13.2%)	0.55	(8.9%)	0.58	(40.3%)	-	
Kernel crossing	return	0.10	(2.9%)	0.20	(3.3%)	-		-	
	syscall	0.10	(2.9%)	0.13	(2.1%)	-		-	
Total		3.36	($\sigma = 0.66$)	6.19	($\sigma = 0.82$)	1.44	($\sigma < 0.01$)	0.38	($\sigma < 0.01$)

Table 1: Sources of packet processing overhead in Linux and Arrakis. All times are averages over 1,000 samples, given in μs (and standard deviation for totals). Arrakis/P uses the POSIX interface, Arrakis/N uses the native Arrakis interface.

The systems have an Intel X520 (82599-based) 10Gb Ethernet adapter and an Intel MegaRAID RS3DC040 RAID controller with 1GB of flash-backed DRAM in front of a 100GB Intel DC S3700 SSD. All machines are connected to a 10Gb Dell PowerConnect 8024F Ethernet switch. One system (the *server*) executes the application under scrutiny, while the others act as clients.

2.1 Networking Stack Overheads

Consider a UDP echo server implemented as a Linux process. The server performs **recvmsg** and **sendmsg** calls in a loop, with no application-level processing, so it stresses packet processing in the OS. Figure 1 depicts the typical workflow for such an application. As Table 1 shows, operating system overhead for packet processing falls into four main categories.

- **Network stack costs**: packet processing at the hardware, IP, and UDP layers.
- **Scheduler overhead**: waking up a process (if necessary), selecting it to run, and context switching to it.
- **Kernel crossings**: from kernel to user space and back.
- **Copying of packet data**: from the kernel to a user buffer on receive, and back on send.

Of the total 3.36 μs (see Table 1) spent processing each packet in Linux, nearly 70% is spent in the network stack. This work is mostly software demultiplexing, security checks, and overhead due to indirection at various layers. The kernel must validate the header of incoming packets and perform security checks on arguments provided by the application when it sends a packet. The stack also performs checks at layer boundaries.

Scheduler overhead depends significantly on whether the receiving process is currently running. If it is, only 5% of processing time is spent in the scheduler; if it is not, the time to context-switch to the server process from the idle process adds an extra 2.2 μs and a further 0.6 μs slowdown in other parts of the network stack.

Cache and lock contention issues on multicore systems add further overhead and are exacerbated by the fact that incoming messages can be delivered on different queues by the network card, causing them to be processed by different CPU cores—which may not be the same as the cores on which the user-level process is scheduled, as depicted in Figure 1. Advanced hardware support such as accelerated receive flow steering [4] aims to mitigate this cost, but these solutions themselves impose non-trivial setup costs [46].

By leveraging hardware support to remove kernel mediation from the data plane, Arrakis can eliminate certain categories of overhead entirely, and minimize the effect of others. Table 1 also shows the corresponding overhead for two variants of Arrakis. Arrakis eliminates scheduling and kernel crossing overhead entirely, because packets are delivered directly to user space. Network stack processing is still required, of course, but it is greatly simplified: it is no longer necessary to demultiplex packets for different applications, and the user-level network stack need not validate parameters provided by the user as extensively as a kernel implementation must. Because each application has a separate network stack, and packets are delivered to cores where the application is running, lock contention and cache effects are reduced.

In the Arrakis network stack, the time to copy packet data to and from user-provided buffers dominates the processing cost, a consequence of the mismatch between the POSIX interface (Arrakis/P) and NIC packet queues. Arriving data is first placed by the network hardware into a network buffer and then copied into the location specified by the POSIX read call. Data to be transmitted is moved into a buffer that can be placed in the network hardware queue; the POSIX write can then return, allowing the user memory to be reused before the data is sent. Although researchers have investigated ways to eliminate this copy from kernel network stacks [45], as Table 1 shows, most of the overhead for a kernel-resident network stack is elsewhere. Once the overhead of traversing the kernel is

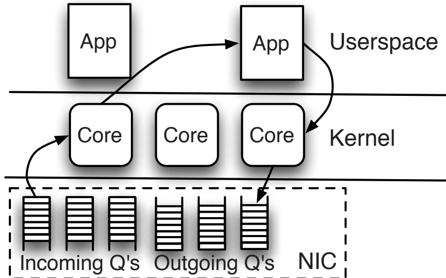


Figure 1: Linux networking architecture and workflow.

removed, there is an opportunity to rethink the POSIX API for more streamlined networking. In addition to a POSIX compatible interface, Arrakis provides a native interface (Arrakis/N) which supports true zero-copy I/O.

2.2 Storage Stack Overheads

To illustrate the overhead of today's OS storage stacks, we conduct an experiment, where we execute small write operations immediately followed by an fsync[1] system call in a tight loop of 10,000 iterations, measuring each operation's latency. We store the file system on a RAM disk, so the measured latencies represent purely CPU overhead.

The overheads shown in Figure 2 stem from data copying between user and kernel space, parameter and access control checks, block and inode allocation, virtualization (the VFS layer), snapshot maintenance (btrfs), as well as metadata updates, in many cases via a journal [53].

While historically these CPU overheads have been insignificant compared to disk access time, recent hardware trends have drastically reduced common-case write storage latency by introducing flash-backed DRAM onto the device. In these systems, OS storage stack overhead becomes a major factor. We measured average write latency to our RAID cache to be 25 μs. PCIe-attached flash storage adapters, like Fusion-IO's ioDrive2, report hardware access latencies as low as 15 μs [24]. In comparison, OS storage stack overheads are high, adding between 40% and 200% for the extended file systems, depending on journal use, and up to 5$\times$ for btrfs. The large standard deviation for btrfs stems from its highly threaded design, used to flush non-critical file system metadata and update reference counts in the background.

2.3 Application Overheads

What do these I/O stack overheads mean to operation latencies within a typical datacenter application? Consider the Redis [18] NoSQL store. Redis persists each write via an operational log (called *append-only file*)[2] and serves reads from an in-memory data structure.

To serve a read, Redis performs a series of operations: First, `epoll` is called to await data for reading, followed

[1] We also tried fdatasync, with negligible difference in latency.

[2] Redis also supports snapshot persistence because of the high per-operation overhead imposed by Linux.

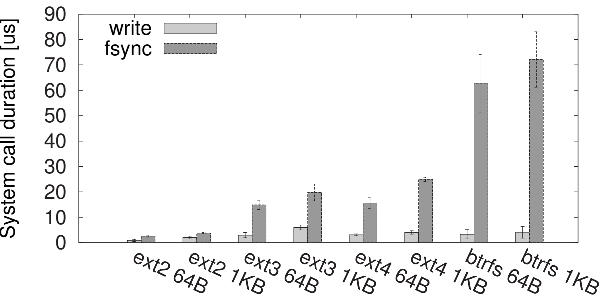


Figure 2: Average overhead in μs of various Linux file system implementations, when conducting small, persistent writes. Error bars show standard deviation.

by `recv` to receive a request. After receiving, the (textual) request is parsed and the key looked-up in memory. Once found, a response is prepared and then, after `epoll` is called again to check whether the socket is ready, sent to the client via `send`. For writes, Redis additionally marshals the operation into log format, `write`s the log and waits for persistence (via the `fsync` call) before responding. Redis also spends time in accounting, access checks, and connection handling (**Other** row in Table 2).

Table 2 shows that a total of 76% of the latency in an average read hit on Linux is due to socket operations. In Arrakis, we reduce socket operation latency by 68%. Similarly, 90% of the latency of a write on Linux is due to I/O operations. In Arrakis we reduce I/O latency by 82%.

We can also see that Arrakis reduces some application-level overheads. This is due to better cache behavior of the user-level I/O stacks and the control/data plane separation evading all kernel crossings. Arrakis' write latency is still dominated by storage access latency (25μs in our system). We expect the gap between Linux and Arrakis performance to widen as faster storage devices appear on the market.

2.4 Hardware I/O Virtualization

Single-Root I/O Virtualization (SR-IOV) [38] is a hardware technology intended to support high-speed I/O for multiple virtual machines sharing a single physical machine. An SR-IOV-capable I/O adapter appears on the PCIe interconnect as a single "physical function" (PCI parlance for a device) which can in turn dynamically create additional "virtual functions". Each of these resembles a PCI device, which can be directly mapped into a different virtual machine and access can be protected via IOMMU (e.g. Intel's VT-d [34]). To the guest operating system, each virtual function can be programmed as if it was a regular physical device, with a normal device driver and an unchanged I/O stack. Hypervisor software with access to the physical hardware (such as Domain 0 in a Xen [9] installation) creates and deletes these virtual functions, and configures filters in the SR-IOV adapter to demultiplex hardware operations to different virtual functions and therefore different guest operating systems.

	Read hit				Durable write			
	Linux		Arrakis/P		Linux		Arrakis/P	
`epoll`	2.42	(27.91%)	1.12	(27.52%)	2.64	(1.62%)	1.49	(4.73%)
`recv`	0.98	(11.30%)	0.29	(7.13%)	1.55	(0.95%)	0.66	(2.09%)
Parse input	0.85	(9.80%)	0.66	(16.22%)	2.34	(1.43%)	1.19	(3.78%)
Lookup/set key	0.10	(1.15%)	0.10	(2.46%)	1.03	(0.63%)	0.43	(1.36%)
Log marshaling	-		-		3.64	(2.23%)	2.43	(7.71%)
`write`	-		-		6.33	(3.88%)	0.10	(0.32%)
`fsync`	-		-		137.84	(84.49%)	24.26	(76.99%)
Prepare response	0.60	(6.92%)	0.64	(15.72%)	0.59	(0.36%)	0.10	(0.32%)
`send`	3.17	(36.56%)	0.71	(17.44%)	5.06	(3.10%)	0.33	(1.05%)
Other	0.55	(6.34%)	0.46	(11.30%)	2.12	(1.30%)	0.52	(1.65%)
Total	8.67	($\sigma = 2.55$)	4.07	($\sigma = 0.44$)	163.14	($\sigma = 13.68$)	31.51	($\sigma = 1.91$)
99th percentile	15.21		4.25		188.67		35.76	

Table 2: Overheads in the Redis NoSQL store for memory reads (hits) and durable writes (legend in Table 1).

In Arrakis, we use SR-IOV, the IOMMU, and supporting adapters to provide direct application-level access to I/O devices. This is a modern implementation of an idea which was implemented twenty years ago with U-Net [54], but generalized to flash storage and Ethernet network adapters. To make user-level I/O stacks tractable, we need a hardware-independent device model and API that captures the important features of SR-IOV adapters [31, 40, 41, 51]; a hardware-specific device driver matches our API to the specifics of the particular device. We discuss this model in the next section, along with potential improvements to the existing hardware to better support user-level I/O.

Remote Direct Memory Access (RDMA) is another popular model for user-level networking [48]. RDMA gives applications the ability to read from or write to a region of virtual memory on a remote machine directly from user-space, bypassing the operating system kernel on both sides. The intended use case is for a parallel program to be able to directly read and modify its data structures even when they are stored on remote machines.

While RDMA provides the performance benefits of user-level networking to parallel applications, it is challenging to apply the model to a broader class of client-server applications [21]. Most importantly, RDMA is point-to-point. Each participant receives an authenticator providing it permission to remotely read/write a particular region of memory. Since clients in client-server computing are not mutually trusted, the hardware would need to keep a separate region of memory for each active connection. Therefore we do not consider RDMA operations here.

3 Design and Implementation

Arrakis has the following design goals:

- **Minimize kernel involvement for data-plane operations:** Arrakis is designed to limit or remove kernel mediation for most I/O operations. I/O requests are routed to and from the application's address space without requiring kernel involvement and without sacrificing security and isolation properties.
- **Transparency to the application programmer:** Arrakis is designed to significantly improve performance without requiring modifications to applications written to the POSIX API. Additional performance gains are possible if the developer can modify the application.
- **Appropriate OS/hardware abstractions:** Arrakis' abstractions should be sufficiently flexible to efficiently support a broad range of I/O patterns, scale well on multicore systems, and support application requirements for locality and load balance.

In this section, we show how we achieve these goals in Arrakis. We describe an ideal set of hardware facilities that should be present to take full advantage of this architecture, and we detail the design of the control plane and data plane interfaces that we provide to the application. Finally, we describe our implementation of Arrakis based on the Barrelfish operating system.

3.1 Architecture Overview

Arrakis targets I/O hardware with support for virtualization, and Figure 3 shows the overall architecture. In this paper, we focus on hardware that can present multiple instances of itself to the operating system and the applications running on the node. For each of these virtualized device instances, the underlying physical device provides unique memory mapped register files, descriptor queues, and interrupts, hence allowing the control plane to map each device instance to a separate protection domain. The device exports a management interface that is accessible from the control plane in order to create or destroy virtual device instances, associate individual instances with network flows or storage areas, and allocate shared resources to the different instances. Applications conduct I/O

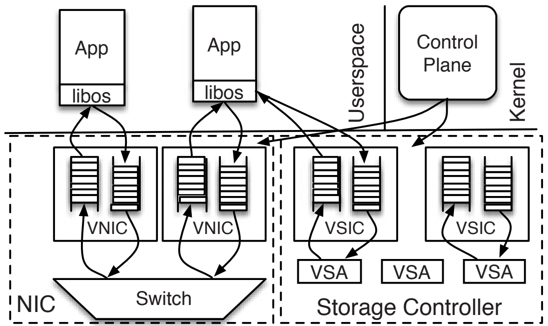


Figure 3: Arrakis architecture. The storage controller maps VSAs to physical storage.

through their protected virtual device instance without requiring kernel intervention. In order to perform these operations, applications rely on a user-level I/O stack that is provided as a library. The user-level I/O stack can be tailored to the application as it can assume exclusive access to a virtualized device instance, allowing us to remove any features not necessary for the application's functionality. Finally, (de-)multiplexing operations and security checks are not needed in this dedicated environment and can be removed.

The user naming and protection model is unchanged. A global naming system is provided by the control plane. This is especially important for sharing stored data. Applications implement their own storage, while the control plane manages naming and coarse-grain allocation, by associating each application with the directories and files it manages. Other applications can still read those files by indirecting through the kernel, which hands the directory or read request to the appropriate application.

3.2 Hardware Model

A key element of our work is to develop a hardware-independent layer for virtualized I/O—that is, a device model providing an "ideal" set of hardware features. This device model captures the functionality required to implement in hardware the data plane operations of a traditional kernel. Our model resembles what is already provided by some hardware I/O adapters; we hope it will provide guidance as to what is needed to support secure user-level networking and storage.

In particular, we assume our network devices provide support for virtualization by presenting themselves as multiple *virtual network interface cards (VNICs)* and that they can also multiplex/demultiplex packets based on complex filter expressions, directly to queues that can be managed entirely in user space without the need for kernel intervention. Similarly, each storage controller exposes multiple *virtual storage interface controllers* (VSICs) in our model. Each VSIC provides independent storage command queues (e.g., of SCSI or ATA format) that are multiplexed by the hardware. Associated with each such virtual interface card (VIC) are *queues* and *rate limiters*. VNICs also provide *filters* and VSICs provide *virtual storage areas*. We discuss these components below.

Queues: Each VIC contains multiple pairs of DMA queues for user-space send and receive. The exact form of these VIC queues could depend on the specifics of the I/O interface card. For example, it could support a scatter/gather interface to aggregate multiple physically-disjoint memory regions into a single data transfer. For NICs, it could also optionally support hardware checksum offload and TCP segmentation facilities. These features enable I/O to be handled more efficiently by performing additional work in hardware. In such cases, the Arrakis system offloads operations and further reduces overheads.

Transmit and receive filters: A transmit filter is a predicate on network packet header fields that the hardware will use to determine whether to send the packet or discard it (possibly signaling an error either to the application or the OS). The transmit filter prevents applications from spoofing information such as IP addresses and VLAN tags and thus eliminates kernel mediation to enforce these security checks. It can also be used to limit an application to communicate with only a pre-selected set of nodes.

A receive filter is a similar predicate that determines which packets received from the network will be delivered to a VNIC and to a specific queue associated with the target VNIC. For example, a VNIC can be set up to receive all packets sent to a particular port, so both connection setup and data transfers can happen at user-level. Installation of transmit and receive filters are privileged operations performed via the kernel control plane.

Virtual storage areas: Storage controllers need to provide an interface via their physical function to map *virtual storage areas* (VSAs) to extents of physical drives, and associate them with VSICs. A typical VSA will be large enough to allow the application to ignore the underlying multiplexing—e.g., multiple erasure blocks on flash, or cylinder groups on disk. An application can store multiple sub-directories and files in a single VSA, providing precise control over multi-object serialization constraints.

A VSA is thus a persistent segment [13]. Applications reference blocks in the VSA using virtual offsets, converted by hardware into physical storage locations. A VSIC may have multiple VSAs, and each VSA may be mapped into multiple VSICs for interprocess sharing.

Bandwidth allocators: This includes support for resource allocation mechanisms such as rate limiters and pacing/traffic shaping of I/O. Once a frame has been removed from a transmit rate-limited or paced queue, the next time another frame could be fetched from that queue is regulated by the rate limits and the inter-packet pacing controls associated with the queue. Installation of these controls are also privileged operations.

In addition, we assume that the I/O device driver supports an introspection interface allowing the control plane to query for resource limits (e.g., the number of queues) and check for the availability of hardware support for I/O processing (e.g., checksumming or segmentation).

Network cards that support SR-IOV have the key elements of this model: they allow the creation of multiple VNICs that each may have multiple send and receive queues, and support at least rudimentary transmit and receive filters. Not all NICs provide the rich filtering semantics we desire; for example, the Intel 82599 can filter only based on source or destination MAC addresses and VLAN tags, not arbitrary predicates on header fields. However, this capability is within reach: some network cards (e.g., Solarflare 10Gb adapters) can already filter packets on all header fields, and the hardware support required for more general VNIC transmit and receive filtering is closely related to that used for techniques like Receive-Side Scaling, which is ubiquitous in high-performance network cards.

Storage controllers have some parts of the technology needed to provide the interface we describe. For example, RAID adapters have a translation layer that is able to provide virtual disks above physical extents, and SSDs use a flash translation layer for wear-leveling. SCSI host-bus adapters support SR-IOV technology for virtualization [40, 41] and can expose multiple VSICs, and the NVMe standard proposes multiple command queues for scalability [35]. Only the required protection mechanism is missing. We anticipate VSAs to be allocated in large chunks and thus hardware protection mechanisms can be coarse-grained and lightweight.

Finally, the number of hardware-supported VICs might be limited. The 82599 [31] and SAS3008 [41] support 64. This number is adequate with respect to the capabilities of the rest of the hardware (e.g., the number of CPU cores), but we expect it to rise. The PCI working group has already ratified an addendum to SR-IOV that increases the supported number of virtual functions to 2048. Bandwidth allocation within the 82599 is limited to weighted round-robin scheduling and rate limiting of each of the 128 transmit/receive queues. Recent research has demonstrated that precise rate limiting in hardware can scale to tens of thousands of traffic classes, enabling sophisticated bandwidth allocation policies [47].

Arrakis currently assumes hardware that can filter and demultiplex flows at a level (packet headers, etc.) corresponding roughly to a traditional OS API, but no higher. An open question is the extent to which hardware that can filter on application-level properties (including content) would provide additional performance benefits.

3.3 VSIC Emulation

To validate our model given limited support from storage devices, we developed prototype VSIC support by dedicating a processor core to emulate the functionality we expect from hardware. The same technique can be used to run Arrakis on systems without VNIC support.

To handle I/O requests from the OS, our RAID controller provides one request and one response descriptor queue of fixed size, implemented as circular buffers along with a software-controlled register (PR) pointing to the head of the request descriptor queue. Request descriptors (RQDs) have a size of 256 bytes and contain a SCSI command, a scatter-gather array of system memory ranges, and a target logical disk number. The SCSI command specifies the type of operation (read or write), total transfer size and on-disk base logical block address (LBA). The scatter-gather array specifies the request's corresponding regions in system memory. Response descriptors refer to completed RQDs by their queue entry and contain a completion code. An RQD can be reused only after its response is received.

We replicate this setup for each VSIC by allocating queue pairs and register files of the same format in system memory mapped into applications and to a dedicated VSIC core. Like the 82599, we limit the maximum number of VSICs to 64. In addition, the VSIC core keeps an array of up to 4 VSA mappings for each VSIC that is programmable only from the control plane. The mappings contain the size of the VSA and an LBA offset within a logical disk, effectively specifying an extent.

In the steady state, the VSIC core polls each VSIC's PR and the latest entry of the response queue of the physical controller in a round-robin fashion. When a new RQD is posted via PR_i on VSIC i, the VSIC core interprets the RQD's logical disk number n as a VSA mapping entry and checks whether the corresponding transfer fits within that VSA's boundaries (i.e., $\text{RQD.LBA} + \text{RQD.size} \leq \text{VSA}_n\text{.size}$). If so, the core copies the RQD to the physical controller's queue, adding VSA_n.offset to RQD.LBA, and sets an unused RQD field to identify the corresponding RQD in the source VSIC before updating the controller's PR register. Upon a response from the controller, the VSIC core copies the response to the corresponding VSIC response queue.

We did not consider VSIC interrupts in our prototype. They can be supported via inter-processor interrupts. To support untrusted applications, our prototype has to translate virtual addresses. This requires it to traverse application page tables for each entry in an RQD's scatter-gather array. In a real system, the IOMMU carries out this task.

On a microbenchmark of 10,000 fixed size write operations of 1KB via a single VSIC to a single VSA, the average overhead of the emulation is 3μs. Executing virtualization code takes 1μs on the VSIC core; the other 2μs are due to cache overheads that we did not quantify further. To measure the expected VSIC performance with direct hardware support, we map the single RAID hardware VSIC directly into the application memory; we report those results in §4.

3.4 Control Plane Interface

The interface between an application and the Arrakis control plane is used to request resources from the system and direct I/O flows to and from user programs. The key abstractions presented by this interface are VICs, doorbells, filters, VSAs, and rate specifiers.

An application can create and delete VICs, and associate *doorbells* with particular events on particular VICs. A doorbell is an IPC end-point used to notify the application that an event (e.g. packet arrival or I/O completion) has occurred, and is discussed below. VICs are hardware resources and so Arrakis must allocate them among applications according to an OS policy. Currently this is done on a first-come-first-served basis, followed by spilling to software emulation (§3.3).

Filters have a *type* (transmit or receive) and a *predicate* which corresponds to a convex sub-volume of the packet header space (for example, obtained with a set of mask-and-compare operations). Filters can be used to specify ranges of IP addresses and port numbers associated with valid packets transmitted/received at each VNIC. Filters are a better abstraction for our purposes than a conventional connection identifier (such as a TCP/IP 5-tuple), since they can encode a wider variety of communication patterns, as well as subsuming traditional port allocation and interface specification.

For example, in the "map" phase of a MapReduce job we would like the application to send to, and receive from, an entire class of machines using the same communication end-point, but nevertheless isolate the data comprising the shuffle from other data. As a second example, web servers with a high rate of incoming TCP connections can run into scalability problems processing connection requests [46]. In Arrakis, a single filter can safely express both a listening socket and all subsequent connections to that socket, allowing server-side TCP connection establishment to avoid kernel mediation.

Applications create a filter with a control plane operation. In the common case, a simple higher-level wrapper suffices: **filter = create_filter(flags, peerlist, servicelist)**. **flags** specifies the filter direction (transmit or receive) and whether the filter refers to the Ethernet, IP, TCP, or UDP header. **peerlist** is a list of accepted communication peers specified according to the filter type, and **servicelist** contains a list of accepted service addresses (e.g., port numbers) for the filter. Wildcards are permitted.

The call to **create_filter** returns **filter**, a kernel-protected capability conferring authority to send or receive packets matching its predicate, and which can then be **assign**ed to a specific queue on a VNIC. VSAs are acquired and assigned to VSICs in a similar fashion.

Finally, a rate specifier can also be assigned to a queue, either to throttle incoming traffic (in the network receive case) or pace outgoing packets and I/O requests. Rate specifiers and filters associated with a VIC queue can be updated dynamically, but all such updates require mediation from the Arrakis control plane.

Our network filters are less expressive than OpenFlow matching tables, in that they do not support priority-based overlapping matches. This is a deliberate choice based on hardware capabilities: NICs today only support simple matching, and to support priorities in the API would lead to unpredictable consumption of hardware resources below the abstraction. Our philosophy is therefore to support expressing such policies only when the hardware can implement them efficiently.

3.5 File Name Lookup

A design principle in Arrakis is to separate file naming from implementation. In a traditional system, the fully-qualified filename specifies the file system used to store the file and thus its metadata format. To work around this, many applications build their own metadata indirection inside the file abstraction [28]. Instead, Arrakis provides applications direct control over VSA storage allocation: an application is free to use its VSA to store metadata, directories, and file data. To allow other applications access to its data, an application can export file and directory names to the kernel virtual file system (VFS). To the rest of the VFS, an application-managed file or directory appears like a remote mount point—an indirection to a file system implemented elsewhere. Operations within the file or directory are handled locally, without kernel intervention.

Other applications can gain access to these files in three ways. By default, the Arrakis application library managing the VSA exports a file server interface; other applications can use normal POSIX API calls via user-level RPC to the embedded library file server. This library can also run as a standalone process to provide access when the original application is not active. Just like a regular mounted file system, the library needs to implement only functionality required for file access on its VSA and may choose to skip any POSIX features that it does not directly support.

Second, VSAs can be mapped into multiple processes. If an application, like a virus checker or backup system, has both permission to read the application's metadata and the appropriate library support, it can directly access the file data in the VSA. In this case, access control is done for the entire VSA and not per file or directory. Finally, the user can direct the originating application to export its data into a standard format, such as a PDF file, stored as a normal file in the kernel-provided file system.

The combination of VFS and library code implement POSIX semantics seamlessly. For example, if execute rights are revoked from a directory, the VFS prevents future traversal of that directory's subtree, but existing RPC connections to parts of the subtree may remain intact until closed. This is akin to a POSIX process retaining a

subdirectory as the current working directory—relative traversals are still permitted.

3.6 Network Data Plane Interface

In Arrakis, applications send and receive network packets by directly communicating with hardware. The data plane interface is therefore implemented in an application library, allowing it to be co-designed with the application [43]. The Arrakis library provides two interfaces to applications. We describe the native Arrakis interface, which departs slightly from the POSIX standard to support true zero-copy I/O; Arrakis also provides a POSIX compatibility layer that supports unmodified applications.

Applications send and receive packets on queues, which have previously been assigned filters as described above. While filters can include IP, TCP, and UDP field predicates, Arrakis does not require the hardware to perform protocol processing, only multiplexing. In our implementation, Arrakis provides a user-space network stack above the data plane interface. This stack is designed to maximize both latency and throughput. We maintain a clean separation between three aspects of packet transmission and reception.

Firstly, packets are transferred asynchronously between the network and main memory using conventional DMA techniques using rings of packet buffer descriptors.

Secondly, the application transfers ownership of a transmit packet to the network hardware by enqueuing a chain of buffers onto the hardware descriptor rings, and acquires a received packet by the reverse process. This is performed by two VNIC driver functions. **send_packet(queue, packet_array)** sends a packet on a queue; the packet is specified by the scatter-gather array **packet_array**, and must conform to a filter already associated with the queue. **receive_packet(queue) = packet** receives a packet from a queue and returns a pointer to it. Both operations are asynchronous. **packet_done(packet)** returns ownership of a received packet to the VNIC.

For optimal performance, the Arrakis stack would interact with the hardware queues not through these calls but directly via compiler-generated, optimized code tailored to the NIC descriptor format. However, the implementation we report on in this paper uses function calls to the driver.

Thirdly, we handle asynchronous notification of events using doorbells associated with queues. Doorbells are delivered directly from hardware to user programs via hardware virtualized interrupts when applications are running and via the control plane to invoke the scheduler when applications are not running. In the latter case, higher latency is tolerable. Doorbells are exposed to Arrakis programs via regular event delivery mechanisms (e.g., a file descriptor event) and are fully integrated with existing I/O multiplexing interfaces (e.g., `select`). They are useful both to notify an application of general availability of packets in receive queues, as well as a lightweight notification mechanism for I/O completion and the reception of packets in high-priority queues.

This design results in a protocol stack that decouples hardware from software as much as possible using the descriptor rings as a buffer, maximizing throughput and minimizing overhead under high packet rates, yielding low latency. On top of this native interface, Arrakis provides POSIX-compatible sockets. This compatibility layer allows Arrakis to support unmodified Linux applications. However, we show that performance gains can be achieved by using the asynchronous native interface.

3.7 Storage Data Plane Interface

The low-level storage API provides a set of commands to asynchronously read, write, and flush hardware caches at any offset and of arbitrary size in a VSA via a command queue in the associated VSIC. To do so, the caller provides an array of virtual memory ranges (address and size) in RAM to be read/written, the VSA identifier, queue number, and matching array of ranges (offset and size) within the VSA. The implementation enqueues the corresponding commands to the VSIC, coalescing and reordering commands if this makes sense to the underlying media. I/O completion events are reported using doorbells. On top of this, a POSIX-compliant file system is provided.

We have also designed a library of *persistent data structures*, Caladan, to take advantage of low-latency storage devices. Persistent data structures can be more efficient than a simple read/write interface provided by file systems. Their drawback is a lack of backwards-compatibility to the POSIX API. Our design goals for persistent data structures are that (1) operations are immediately persistent, (2) the structure is robust versus crash failures, and (3) operations have minimal latency.

We have designed persistent log and queue data structures according to these goals and modified a number of applications to use them (e.g., §4.4). These data structures manage all metadata required for persistence, which allows tailoring of that data to reduce latency. For example, metadata can be allocated along with each data structure entry and persisted in a single hardware write operation. For the log and queue, the only metadata that needs to be kept is where they start and end. Pointers link entries to accommodate wrap-arounds and holes, optimizing for linear access and efficient prefetch of entries. By contrast, a filesystem typically has separate inodes to manage block allocation. The in-memory layout of Caladan structures is as stored, eliminating marshaling.

The log API includes operations to open and close a log, create log entries (for metadata allocation), append them to the log (for persistence), iterate through the log (for reading), and trim the log. The queue API adds a `pop` operation to combine trimming and reading the queue. Persistence is asynchronous: an append operation returns immediately

with a callback on persistence. This allows us to mask remaining write latencies, e.g., by optimistically preparing network responses to clients, while an entry is persisted.

Entries are allocated in multiples of the storage hardware's minimum transfer unit (MTU—512 bytes for our RAID controller, based on SCSI) and contain a header that denotes the true (byte-granularity) size of the entry and points to the offset of the next entry in a VSA. This allows entries to be written directly from memory, without additional marshaling. At the end of each entry is a marker that is used to determine whether an entry was fully written (empty VSA space is always zero). By issuing appropriate cache flush commands to the storage hardware, Caladan ensures that markers are written after the rest of the entry (cf. [17]).

Both data structures are identified by a header at the beginning of the VSA that contains a version number, the number of entries, the MTU of the storage device, and a pointer to the beginning and end of the structure within the VSA. Caladan repairs a corrupted or outdated header lazily in the background upon opening, by looking for additional, complete entries from the purported end of the structure.

3.8 Implementation

The Arrakis operating system is based upon a fork of the Barrelfish [10] multicore OS code base [1]. We added 33,786 lines of code to the Barrelfish code base in order to implement Arrakis. Barrelfish lends itself well to our approach, as it already provides a library OS. We could have also chosen to base Arrakis on the Xen [9] hypervisor or the Intel Data Plane Development Kit (DPDK) [32] running on Linux; both provide user-level access to the network interface via hardware virtualization. However, implementing a library OS from scratch on top of a monolithic OS would have been more time consuming than extending the Barrelfish library OS.

We extended Barrelfish with support for SR-IOV, which required modifying the existing PCI device manager to recognize and handle SR-IOV extended PCI capabilities. We implemented a physical function driver for the Intel 82599 10G Ethernet Adapter [31] that can initialize and manage a number of virtual functions. We also implemented a virtual function driver for the 82599, including support for Extended Message Signaled Interrupts (MSI-X), which are used to deliver per-VNIC doorbell events to applications. Finally, we implemented drivers for the Intel IOMMU [34] and the Intel RS3 family of RAID controllers [33]. In addition—to support our benchmark applications—we added several POSIX APIs that were not implemented in the Barrelfish code base, such as POSIX threads, many functions of the POSIX sockets API, as well as the epoll interface found in Linux to allow scalable polling of a large number of file descriptors. Barrelfish already supports standalone user-mode device drivers, akin to those found in microkernels. We created shared library versions of the drivers, which we link to each application.

We have developed our own user-level network stack, Extaris. Extaris is a shared library that interfaces directly with the virtual function device driver and provides the POSIX sockets API and Arrakis's native API to the application. Extaris is based in part on the low-level packet processing code of the lwIP network stack [42]. It has identical capabilities to lwIP, but supports hardware offload of layer 3 and 4 checksum operations and does not require any synchronization points or serialization of packet operations. We have also developed our own storage API layer, as described in §3.7 and our library of persistent data structures, Caladan.

3.9 Limitations and Future Work

Due to the limited filtering support of the 82599 NIC, our implementation uses a different MAC address for each VNIC, which we use to direct flows to applications and then do more fine-grain filtering in software, within applications. The availability of more general-purpose filters would eliminate this software overhead.

Our implementation of the virtual function driver does not currently support the "transmit descriptor head writeback" feature of the 82599, which reduces the number of PCI bus transactions necessary for transmit operations. We expect to see a 5% network performance improvement from adding this support.

The RS3 RAID controller we used in our experiments does not support SR-IOV or VSAs. Hence, we use its physical function, which provides one hardware queue, and we map a VSA to each logical disk provided by the controller. We still use the IOMMU for protected access to application virtual memory, but the controller does not protect access to logical disks based on capabilities. Our experience with the 82599 suggests that hardware I/O virtualization incurs negligible performance overhead versus direct access to the physical function. We expect this to be similar for storage controllers.

4 Evaluation

We evaluate Arrakis on four cloud application workloads: a typical, read-heavy load pattern observed in many large deployments of the memcached distributed object caching system, a write-heavy load pattern to the Redis persistent NoSQL store, a workload consisting of a large number of individual client HTTP requests made to a farm of web servers via an HTTP load balancer and, finally, the same benchmark via an IP-layer middlebox. We also examine the system under maximum load in a series of microbenchmarks and analyze performance crosstalk among multiple networked applications. Using these experiments, we seek to answer the following questions:

- What are the major contributors to performance overhead in Arrakis and how do they compare to those of Linux (presented in §2)?
- Does Arrakis provide better latency and throughput for real-world cloud applications? How does the throughput scale with the number of CPU cores for these workloads?
- Can Arrakis retain the benefits of user-level application execution and kernel enforcement, while providing high-performance packet-level network IO?
- What additional performance gains are possible by departing from the POSIX interface?

We compare the performance of the following OS configurations: Linux kernel version 3.8 (Ubuntu version 13.04), Arrakis using the POSIX interface (Arrakis/P), and Arrakis using its native interface (Arrakis/N).

We tuned Linux network performance by installing the latest ixgbe device driver version 3.17.3 and disabling receive side scaling (RSS) when applications execute on only one processor. RSS spreads packets over several NIC receive queues, but incurs needless coherence overhead on a single core. The changes yield a throughput improvement of 10% over non-tuned Linux. We use the kernel-shipped MegaRAID driver version 6.600.18.00-rc1.

Linux uses a number of performance-enhancing features of the network hardware, which Arrakis does not currently support. Among these features is the use of direct processor cache access by the NIC, TCP and UDP segmentation offload, large receive offload, and network packet header splitting. All of these features can be implemented in Arrakis; thus, our performance comparison is weighted in favor of Linux.

4.1 Server-side Packet Processing Performance

We load the UDP echo benchmark from §2 on the server and use all other machines in the cluster as load generators. These generate 1 KB UDP packets at a fixed rate and record the rate at which their echoes arrive. Each experiment exposes the server to maximum load for 20 seconds.

Shown in Table 1, compared to Linux, Arrakis eliminates two system calls, software demultiplexing overhead, socket buffer locks, and security checks. In Arrakis/N, we additionally eliminate two socket buffer copies. Arrakis/P incurs a total server-side overhead of 1.44 μs, 57% less than Linux. Arrakis/N reduces this overhead to 0.38 μs.

The echo server is able to add a configurable delay before sending back each packet. We use this delay to simulate additional application-level processing time at the server. Figure 4 shows the average throughput attained by each system over various such delays; the theoretical line rate is 1.26M pps with zero processing.

In the best case (no additional processing time), Arrakis/P achieves 2.3× the throughput of Linux. By departing from POSIX, Arrakis/N achieves 3.9× the throughput of Linux. The relative benefit of Arrakis disappears at 64 μs. To gauge how close Arrakis comes to the maximum possible throughput, we embedded a minimal echo server directly into the NIC device driver, eliminating any remaining API overhead. Arrakis/N achieves 94% of the driver limit.

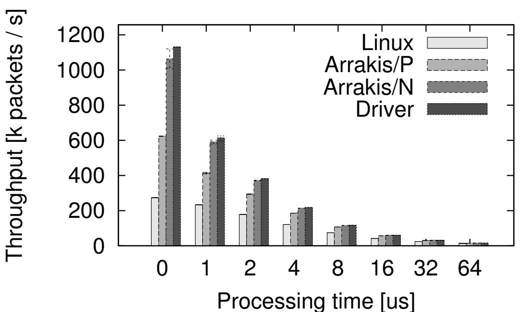


Figure 4: Average UDP echo throughput for packets with 1024 byte payload over various processing times. The top y-axis value shows theoretical maximum throughput on the 10G network. Error bars in this and following figures show min/max measured over 5 repeats of the experiment.

4.2 Memcached Key-Value Store

Memcached is an in-memory key-value store used by many cloud applications. It incurs a processing overhead of 2–3 μs for an average object fetch request, comparable to the overhead of OS kernel network processing.

We benchmark memcached 1.4.15 by sending it requests at a constant rate via its binary UDP protocol, using a tool similar to the popular memslap benchmark [2]. We configure a workload pattern of 90% fetch and 10% store requests on a pre-generated range of 128 different keys of a fixed size of 64 bytes and a value size of 1 KB, in line with real cloud deployments [7].

To measure network stack scalability for multiple cores, we vary the number of memcached server processes. Each server process executes independently on its own port number, such that measurements are not impacted by scalability bottlenecks in memcached itself, and we distribute load equally among the available memcached instances. On Linux, memcached processes share the kernel-level network stack. On Arrakis, each process obtains its own VNIC with an independent set of packet queues, each controlled by an independent instance of Extaris.

Figure 5 shows that memcached on Arrakis/P achieves 1.7× the throughput of Linux on one core, and attains near line-rate at 4 CPU cores. The slightly lower throughput on all 6 cores is due to contention with Barrelfish system management processes [10]. By contrast, Linux throughput nearly plateaus beyond two cores. A single, multi-threaded memcached instance shows no noticeable throughput difference to the multi-process scenario. This is not surprising as memcached is optimized to scale well.

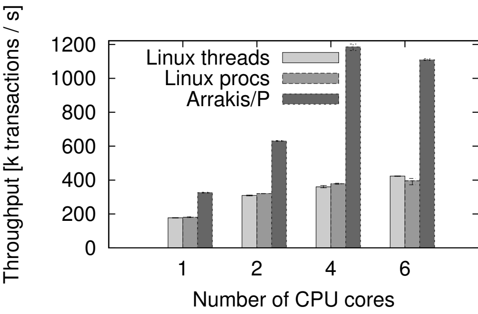


Figure 5: Average memcached transaction throughput and scalability. Top y-axis value = 10Gb/s.

To conclude, the separation of network stack and application in Linux provides only limited information about the application's packet processing and poses difficulty assigning threads to the right CPU core. The resulting cache misses and socket lock contention are responsible for much of the Linux overhead. In Arrakis, the application is in control of the whole packet processing flow: assignment of packets to packet queues, packet queues to cores, and finally the scheduling of its own threads on these cores. The network stack thus does not need to acquire any locks, and packet data is always available in the right processor cache.

Memcached is also an excellent example of the communication endpoint abstraction: we can create hardware filters to allow packet reception and transmission only between the memcached server and a designated list of client machines that are part of the cloud application. In the Linux case, we have to filter connections in the application.

4.3 Arrakis Native Interface Case Study

As a case study, we modified memcached to make use of Arrakis/N. In total, 74 lines of code were changed, with 11 pertaining to the receive side, and 63 to the send side. On the receive side, the changes involve eliminating memcached's receive buffer and working directly with pointers to packet buffers provided by Extaris, as well as returning completed buffers to Extaris. The changes increase average throughput by 9% over Arrakis/P. On the send side, changes include allocating a number of send buffers to allow buffering of responses until fully sent by the NIC, which now must be done within memcached itself. They also involve the addition of reference counts to hash table entries and send buffers to determine when it is safe to reuse buffers and hash table entries that might otherwise still be processed by the NIC. We gain an additional 10% average throughput when using the send side API in addition to the receive side API.

4.4 Redis NoSQL Store

Redis [18] extends the memcached model from a cache to a persistent NoSQL object store. Our results in Table 2 show that Redis operations—while more laborious than Memcached—are still dominated by I/O stack overheads.

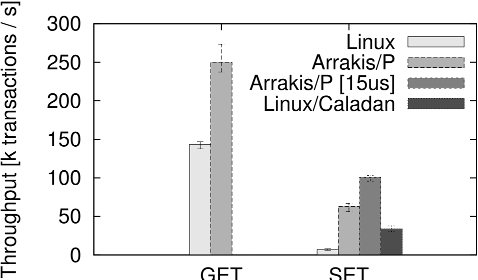


Figure 6: Average Redis transaction throughput for GET and SET operations. The Arrakis/P [15us] and Linux/Caladan configurations apply only to SET operations.

Redis can be used in the same scenario as Memcached and we follow an identical experiment setup, using Redis version 2.8.5. We use the benchmarking tool distributed with Redis and configure it to execute GET and SET requests in two separate benchmarks to a range of 65,536 random keys with a value size of 1,024 bytes, persisting each SET operation individually, with a total concurrency of 1,600 connections from 16 benchmark clients executing on the client machines. Redis is single-threaded, so we investigate only single-core performance.

The Arrakis version of Redis uses Caladan. We changed 109 lines in the application to manage and exchange records with the Caladan log instead of a file. We did not eliminate Redis' marshaling overhead (cf. Table 2). If we did, we would save another 2.43 μs of write latency. Due to the fast I/O stacks, Redis' read performance mirrors that of Memcached and write latency improves by 63%, while write throughput improves vastly, by 9×.

To investigate what would happen if we had access to state-of-the-art storage hardware, we simulate (via a write-delaying RAM disk) a storage backend with 15 μs write latency, such as the ioDrive2 [24]. Write throughput improves by another 1.6×, nearing Linux read throughput.

Both network and disk virtualization is needed for good Redis performance. We tested this by porting Caladan to run on Linux, with the unmodified Linux network stack. This improved write throughput by only 5× compared to Linux, compared to 9× on Arrakis.

Together, the combination of data-plane network and storage stacks can yield large benefits in latency and throughput for both read and write-heavy workloads. The tight integration of storage and data structure in Caladan allows for a number of latency-saving techniques that eliminate marshaling overhead, book-keeping of journals for file system metadata, and can offset storage allocation overhead. These benefits will increase further with upcoming hardware improvements.

4.5 HTTP Load Balancer

To aid scalability of web services, HTTP load balancers are often deployed to distribute client load over a number

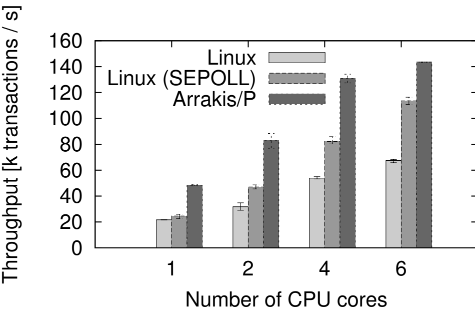


Figure 7: Average HTTP transaction throughput and scalability of haproxy.

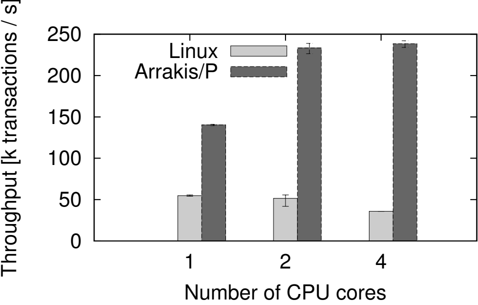


Figure 8: Average HTTP transaction throughput and scalability of the load balancing middlebox. Top y-axis value = 10Gb/s.

of web servers. A popular HTTP load balancer employed by many web and cloud services, such as Amazon EC2 and Twitter, is haproxy [3]. In these settings, many connections are constantly opened and closed and the OS needs to handle the creation and deletion of the associated socket data structures.

To investigate how performance is impacted when many connections need to be maintained, we configure five web servers and one load balancer. To minimize overhead at the web servers, we deploy a simple static web page of 1,024 bytes, served out of main memory.These same web server hosts also serve as workload generators, using ApacheBench version 2.3 to conduct as many concurrent requests for the web page as possible. Each request is encapsulated in its own TCP connection. On the load balancer host, we deploy haproxy version 1.4.24, configured to distribute incoming load in a round-robin fashion. We run multiple copies of the haproxy process on the load balancing node, each executing on their own port number. We configure the ApacheBench instances to distribute their load equally among the available haproxy instances.

Haproxy relies on cookies, which it inserts into the HTTP stream to remember connection assignments to backend web servers under possible client re-connects. This requires it to interpret the HTTP stream for each client request. Linux provides an optimization called TCP splicing that allows applications to forward traffic between two sockets without user-space involvement. This reduces the overhead of kernel crossings when connections are long-lived. We enable haproxy to use this feature on Linux when beneficial.

Finally, haproxy contains a feature known as "speculative epoll" (SEPOLL), which uses knowledge about typical socket operation flows within the Linux kernel to avoid calls to the epoll interface and optimize performance. Since the Extaris implementation differs from that of the Linux kernel network stack, we were not able to use this interface on Arrakis, but speculate that this feature could be ported to Arrakis to yield similar performance benefits. To show the effect of the SEPOLL feature, we repeat the Linux benchmark both with and without it and show both results.

In Figure 7, we can see that Arrakis outperforms Linux in both regular and SEPOLL configurations on a single core, by a factor of 2.2 and 2, respectively. Both systems show similar scalability curves. Note that Arrakis's performance on 6 CPUs is affected by background activity on Barrelfish.

To conclude, connection oriented workloads require a higher number of system calls for setup (`accept` and `setsockopt`) and teardown (`close`). In Arrakis, we can use filters, which require only one control plane interaction to specify which clients and servers may communicate with the load balancer service. Further socket operations are reduced to function calls in the library OS, with lower overhead.

4.6 IP-layer Middlebox

IP-layer middleboxes are ubiquitous in today's wide area networks (WANs). Common middleboxes perform tasks, such as firewalling, intrusion detection, network address translation, and load balancing. Due to the complexity of their tasks, middleboxes can benefit from the programming and run-time convenience provided by an OS through its abstractions for safety and resource management.

We implemented a simple user-level load balancing middlebox using raw IP sockets [5]. Just like haproxy, the middlebox balances an incoming TCP workload to a set of back-end servers. Unlike haproxy, it is operating completely transparently to the higher layer protocols. It simply rewrites source and destination IP addresses and TCP port numbers contained in the packet headers. It monitors active TCP connections and uses a hash table to remember existing connection assignments. Responses by the back-end web servers are also intercepted and forwarded back to the corresponding clients. This is sufficient to provide the same load balancing capabilities as in the haproxy experiment. We repeat the experiment from §4.5, replacing haproxy with our middlebox.

The simpler nature of the middlebox is reflected in the throughput results (see Figure 8). Both Linux and Arrakis perform better. Because the middlebox performs less application-level work than haproxy, performance factors are largely due to OS-level network packet processing. As a consequence, Arrakis' benefits are more prominent,

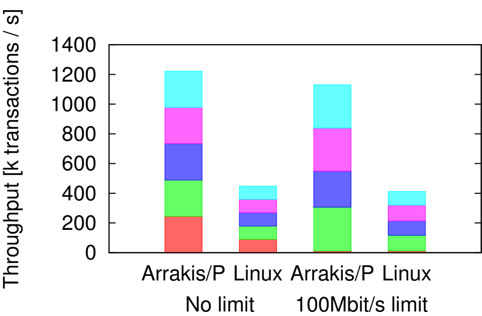


Figure 9: Memcached transaction throughput over 5 instances (colors), with and without rate limiting.

and its performance is 2.6× that of Linux. We also see an interesting effect: the Linux implementation does not scale at all in this configuration. The reason for this are the raw IP sockets, which carry no connection information. Without an indication of which connections to steer to which sockets, each middlebox instance has to look at each incoming packet to determine whether it should handle it. This added overhead outweighs any performance gained via parallelism. In Arrakis, we can configure the hardware filters to steer packets based on packet header information and thus scale until we quickly hit the NIC throughput limit at two cores.

We conclude that Arrakis allows us to retain the safety, abstraction, and management benefits of software development at user-level, while vastly improving the performance of low level packet operations. Filters provide a versatile interface to steer packet workloads based on arbitrary information stored in packet headers to effectively leverage multi-core parallelism, regardless of protocol specifics.

4.7 Performance Isolation

We show that QoS limits can be enforced in Arrakis, by simulating a simple multi-tenant scenario with 5 memcached instances pinned to distinct cores, to minimize processor crosstalk. One tenant has an SLA that allows it to send up to 100Mb/s. The other tenants are not limited.

We use rate specifiers in Arrakis to set the transmit rate limit of the VNIC of the limited process. On Linux, we use queuing disciplines [29] (specifically, HTB [20]) to rate limit the source port of the equivalent process.

We repeat the experiment from §4.2, plotting the throughput achieved by each memcached instance, shown in Figure 9. The bottom-most process (barely visible) is rate-limited to 100Mb/s in the experiment shown on the right hand side of the figure. All runs remained within the error bars shown in Figure 5. When rate-limiting, a bit of the total throughput is lost for both OSes because clients keep sending packets at the same high rate. These consume network bandwidth, even when later dropped due to the rate limit.

We conclude that it is possible to provide the same kind of QoS enforcement—in this case, rate limiting—in Arrakis, as in Linux. Thus, we are able to retain the protection and policing benefits of user-level application execution, while providing improved network performance.

5 Discussion

In this section, we discuss how we can extend the Arrakis model to apply to virtualized guest environments, as well as to interprocessor interrupts.

5.1 Arrakis as Virtualized Guest

Arrakis' model can be extended to virtualized environments. Making Arrakis a host in this environment is straight-forward—this is what the technology was originally designed for. The best way to support Arrakis as a guest is by moving the control plane into the virtual machine monitor (VMM). Arrakis guest applications can then allocate virtual interface cards directly from the VMM. A simple way of accomplishing this is by pre-allocating a number of virtual interface cards in the VMM to the guest and let applications pick only from this pre-allocated set, without requiring a special interface to the VMM.

The hardware limits apply to a virtualized environment in the same way as they do in the regular Arrakis environment. We believe the current limits on virtual adapters (typically 64) to be balanced with the number of available processing resources.

5.2 Virtualized Interprocessor Interrupts

To date, most parallel applications are designed assuming that shared-memory is (relatively) efficient, while interprocessor signaling is (relatively) inefficient. A cache miss to data written by another core is handled in hardware, while alerting a thread on another processor requires kernel mediation on both the sending and receiving side. The kernel is involved even when signaling an event between two threads running inside the same application.

With kernel bypass, a remote cache miss and a remote event delivery are similar in cost at a physical level. Modern hardware already provides the operating system the ability to control how device interrupts are routed. To safely deliver an interrupt within an application, without kernel mediation, requires that the hardware add access control. With this, the kernel could configure the interrupt routing hardware to permit signaling among cores running the same application, trapping to the kernel only when signaling between different applications.

6 Related Work

SPIN [14] and Exokernel [25] reduced shared kernel components to allow each application to have customized operating system management. Nemesis [15] reduces shared components to provide more performance isolation for multimedia applications. All three mediated I/O in the kernel. Relative to these systems, Arrakis shows that

application customization is consistent with very high performance.

Following U-Net, a sequence of hardware standards such as VIA [19] and Infiniband [30] addressed the challenge of minimizing, or eliminating entirely, operating system involvement in sending and receiving network packets in the common case. To a large extent, these systems have focused on the needs of parallel applications for high throughout, low overhead communication. Arrakis supports a more general networking model including client-server and peer-to-peer communication.

Our work was inspired in part by previous work on Dune [11], which used nested paging to provide support for user-level control over virtual memory, and Exitless IPIs [26], which presented a technique to demultiplex hardware interrupts between virtual machines without mediation from the virtual machine monitor.

Netmap [49] implements high throughput network I/O by doing DMAs directly from user space. Sends and receives still require system calls, as the OS needs to do permission checks on every operation. Throughput is achieved at the expense of latency, by batching reads and writes. Similarly, IX [12] implements a custom, per-application network stack in a protected domain accessed with batched system calls. Arrakis eliminates the need for batching by handling operations at user level in the common case.

Concurrently with our work, mTCP uses Intel's DPDK interface to implement a scalable user-level TCP [36]; mTCP focuses on scalable network stack design, while our focus is on the operating system API for general client-server applications. We expect the performance of Extaris and mTCP to be similar. OpenOnload [50] is a hybrid user- and kernel-level network stack. It is completely binary-compatible with existing Linux applications; to support this, it has to keep a significant amount of socket state in the kernel and supports only a traditional socket API. Arrakis, in contrast, allows applications to access the network hardware directly and does not impose API constraints.

Recent work has focused on reducing the overheads imposed by traditional file systems and block device drivers, given the availability of low latency persistent memory. DFS [37] and PMFS [23] are file systems designed for these devices. DFS relies on the flash storage layer for functionality traditionally implemented in the OS, such as block allocation. PMFS exploits the byte-addressability of persistent memory, avoiding the block layer. Both DFS and PMFS are implemented as kernel-level file systems, exposing POSIX interfaces. They focus on optimizing file system and device driver design for specific technologies, while Arrakis investigates how to allow applications fast, customized device access.

Moneta-D [16] is a hardware and software platform for fast, user-level I/O to solid-state devices. The hardware and operating system cooperate to track permissions on hardware extents, while a user-space driver communicates with the device through a virtual interface. Applications interact with the system through a traditional file system. Moneta-D is optimized for large files, since each open operation requires communication with the OS to check permissions; Arrakis does not have this issue, since applications have complete control over their VSAs. Aerie [53] proposes an architecture in which multiple processes communicate with a trusted user-space file system service for file metadata and lock operations, while directly accessing the hardware for reads and data-only writes. Arrakis provides more flexibility than Aerie, since storage solutions can be integrated tightly with applications rather than provided in a shared service, allowing for the development of higher-level abstractions, such as persistent data structures.

7 Conclusion

In this paper, we described and evaluated Arrakis, a new operating system designed to remove the kernel from the I/O data path without compromising process isolation. Unlike a traditional operating system, which mediates all I/O operations to enforce process isolation and resource limits, Arrakis uses device hardware to deliver I/O directly to a customized user-level library. The Arrakis kernel operates in the control plane, configuring the hardware to limit application misbehavior.

To demonstrate the practicality of our approach, we have implemented Arrakis on commercially available network and storage hardware and used it to benchmark several typical server workloads. We are able to show that protection and high performance are not contradictory: end-to-end client read and write latency to the Redis persistent NoSQL store is 2–5× faster and write throughput 9× higher on Arrakis than on a well-tuned Linux implementation.

Acknowledgments

This work was supported by NetApp, Google, and the National Science Foundation. We would like to thank the anonymous reviewers and our shepherd, Emmett Witchel, for their comments and feedback. We also thank Oleg Godunok for implementing the IOMMU driver, Antoine Kaufmann for implementing MSI-X support, and Taesoo Kim for implementing interrupt support into Extaris.

References

[1] `http://www.barrelfish.org/`.

[2] `http://www.libmemcached.org/`.

[3] `http://haproxy.1wt.eu`.

[4] Scaling in the Linux networking stack. `https://www.kernel.org/doc/Documentation/networking/scaling.txt`.

[5] Linux IPv4 raw sockets, May 2012. http://man7.org/linux/man-pages/man7/raw.7.html.

[6] D. Abramson. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, 2006.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[8] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[10] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, 2009.

[11] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*, 2012.

[12] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *OSDI*, 2014.

[13] A. Bensoussan, C. Clingen, and R. Daley. The Multics virtual memory: Concepts and design. *CACM*, 15:308–318, 1972.

[14] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, 1995.

[15] R. Black, P. T. Barham, A. Donnelly, and N. Stratford. Protocol implementation in a vertically structured operating system. In *LCN*, 1997.

[16] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. ASPLOS, 2012.

[17] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *SOSP*, 2013.

[18] Citrusbyte. Redis. http://redis.io/.

[19] Compaq Computer Corp., Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification*, version 1.0 edition, December 1997.

[20] M. Devera. HTB Linux queuing discipline manual – User Guide, May 2002. http://luxik.cdi.cz/~devik/qos/htb/userg.pdf.

[21] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, 2014.

[22] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM*, 1994.

[23] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.

[24] Fusion-IO. *ioDrive2 and ioDrive2 Duo Multi Level Cell*, 2014. Product Datasheet. http://www.fusionio.com/load/-media-/2rezss/docsLibrary/FIO_DS_ioDrive2.pdf.

[25] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on Exokernel systems. *TOCS*, 20(1):49–83, Feb 2002.

[26] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: bare-metal performance for I/O virtualization. In *ASPLOS*, 2012.

[27] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *OSDI*, 2012.

[28] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *SOSP*, 2011.

[29] B. Hubert. Linux advanced routing & traffic control HOWTO. http://www.lartc.org/howto/.

[30] Infiniband Trade Organization. Introduction to Infiniband for end users. https://cw.infinibandta.org/document/dl/7268, April 2010.

[31] Intel Corporation. *Intel 82599 10 GbE Controller Datasheet*, December 2010. Revision 2.6. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[32] Intel Corporation. *Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide*, Aug 2013. Reference Number: 326003-003.

[33] Intel Corporation. *Intel RAID Controllers RS3DC080 and RS3DC040*, Aug 2013. Product Brief. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/raid-controller-rs3dc-brief.pdf.

[34] Intel Corporation. Intel virtualization technology for directed I/O architecture specification. Technical Report Order Number: D51397-006, Intel Corporation, Sep 2013.

[35] Intel Corporation. *NVM Express*, revision 1.1a edition, Sep 2013. http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1a.pdf.

[36] E. Jeong, S. Woo, M. Jamshed, H. J. S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[37] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, Sep 2010.

[38] P. Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. *Intel application note*, 321211–002, Jan 2011.

[39] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J.Sel. A. Commun.*, 14(7):1280–1297, Sep 2006.

[40] LSI Corporation. *LSISAS2308 PCI Express to 8-Port 6Gb/s SAS/SATA Controller*, Feb 2010. Product Brief. http://www.lsi.com/downloads/Public/SAS%20ICs/LSI_PB_SAS2308.pdf.

[41] LSI Corporation. *LSISAS3008 PCI Express to 8-Port 12Gb/s SAS/SATA Controller*, Feb 2014. Product Brief. http://www.lsi.com/downloads/Public/SAS%20ICs/LSI_PB_SAS3008.pdf.

[42] lwIP. http://savannah.nongnu.org/projects/lwip/.

[43] I. Marinos, R. N. M. Watson, and M. Handley. Network stack specialization for performance. In *SIGCOMM*, 2014.

[44] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *OSDI*, 1996.

[45] V. S. Pai, P. Druschel, and W. Zwanepoel. IO-Lite: A unified I/O buffering and caching system. In *OSDI*, 1999.

[46] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *EuroSys*, 2012.

[47] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *NSDI*, 2014.

[48] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.

[49] L. Rizzo. Netmap: A novel framework for fast packet I/O. In *USENIX ATC*, 2012.

[50] SolarFlare Communications, Inc. OpenOnload. http://www.openonload.org/.

[51] Solarflare Communications, Inc. *Solarflare SFN5122F Dual-Port 10GbE Enterprise Server Adapter*, 2010.

[52] A. Trivedi, P. Stuedi, B. Metzler, R. Pletka, B. G. Fitch, and T. R. Gross. Unified high-performance I/O: One stack to rule them all. In *HotOS*, 2013.

[53] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*, 2014.

[54] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *SOSP*, 1995.

Decoupling Cores, Kernels, and Operating Systems

Gerd Zellweger, Simon Gerber, Kornilios Kourtis, Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zurich

Abstract

We present Barrelfish/DC, an extension to the Barrelfish OS which decouples physical cores from a native OS kernel, and furthermore the kernel itself from the rest of the OS and application state. In Barrelfish/DC, native kernel code on any core can be quickly replaced, kernel state moved between cores, and cores added and removed from the system transparently to applications *and* OS processes, which continue to execute.

Barrelfish/DC is a multikernel with two novel ideas: the use of *boot drivers* to abstract cores as regular devices, and a partitioned capability system for memory management which *externalizes core-local kernel state*.

We show by performance measurements of real applications and device drivers that the approach is practical enough to be used for a number of purposes, such as online kernel upgrades, and temporarily delivering hard real-time performance by executing a process under a specialized, single-application kernel.

1 Introduction

The hardware landscape is increasingly dynamic. Future machines will contain large numbers of heterogeneous cores which will be powered on and off individually in response to workload changes. Cores themselves will have porous boundaries: some may be dynamically fused or split to provide more energy-efficient computation. Existing OS designs like Linux and Windows assume a static number of homogeneous cores, with recent extensions to allow core hotplugging.

We present Barrelfish/DC, an OS design based on the principle that all cores are fully dynamic. Barrelfish/DC is based on the Barrelfish research OS [5] and exploits the "multikernel" architecture to separate the OS state for each core. We show that Barrelfish/DC can handle dynamic cores more flexibly and with far less overhead than Linux, and also that the approach brings additional benefits in functionality.

A key challenge with dynamic cores is safely disposing of per-core OS state when removing a core from the system: this process takes time and can dominate the hardware latency of powering the core down, reducing any benefit in energy consumption. Barrelfish/DC addresses this challenge by externalizing all the per-core OS and application state of a system into objects called *OSnodes*, which can be executed lazily on another core. While this general idea has been proposed before (notably, it is used in Chameleon [37] to clean up interrupt state), Barrelfish/DC takes the concept much further in *completely* decoupling the OSnode from the kernel, and this in turn from the physical core.

While transparent to applications, this new design choice implies additional benefits not seen in prior systems: Barrelfish/DC can completely replace the OS kernel code running on any single core or subset of cores in the system at runtime, without disruption to any other OS or application code, including that running on the core. Kernels can be upgraded or bugs fixed without downtime, or replaced temporarily, for example to enable detailed instrumentation, to change a scheduling algorithm, or to provide a different kind of service such as performance-isolated, hard real-time processing for a bounded period.

Furthermore, per-core OS state can be moved between slow, low-power cores and fast, energy-hungry cores. Multiple cores' state can be temporarily aggregated onto a single core to further trade-off performance and power, or to dedicate an entire package to running a single job for a limited period. Parts of Barrelfish/DC can be moved onto and off cores optimized for particular workloads. Cores can be fused [26] transparently, and SMT threads [29, 34] or cores sharing functional units [12] can be selectively used for application threads or OS accelerators.

Barrelfish/DC relies on several innovations which form the main contributions of this paper. Barrelfish/DC treats a CPU core as being a special case of a peripheral device, and introduces the concept of a *boot driver*, which can start, stop, and restart a core while running elsewhere. We

use a *partitioned capability system* for memory management which allows us to completely externalize all OS state for a core. This in turn permits a kernel to be essentially stateless, and easily replaced while Barrelfish/DC continues to run. We factor the OS into per-core kernels[1] and *OSnodes*, and a *Kernel Control Block* provides a kernel-readable handle on the total state of an OSnode.

In the next section, we lay out the recent trends in hardware design and software requirements that motivate the ideas in Barrelfish/DC. Following this, in Section 3 we discuss in more detail the background to our work, and related systems and techniques. In Section 4 we present the design of Barrelfish/DC, in particular the key ideas mentioned above. In Section 5 we show by means of microbenchmarks and real applications (a web server and the PostgreSQL database) that the new functionality of Barrelfish/DC incurs negligible overhead, as well as demonstrating how Barrelfish/DC can provide worst-case execution time guarantees for applications by temporarily isolating cores. Finally, we discuss Barrelfish/DC limitations and future work in Section 6, and conclude in Section 7.

2 Motivation and Background

Barrelfish/DC fully decouples cores from kernels (supervisory programs running in kernel mode), and moreover both of them from the per-core state of the OS as a whole and its associated applications (threads, address spaces, communication channels, etc.). This goes considerably beyond the core hotplug or dynamic core support in today's OSes. Figure 1 shows the range of primitive kernel operations that Barrelfish/DC supports transparently to applications and without downtime as the system executes:

- A kernel on a core can be rebooted or replaced.
- The per-core OS state can be moved between cores.
- Multiple per-core OS components can be relocated to temporarily "share" a core.

In this section we argue why such functionality will become important in the future, based on recent trends in hardware and software.

2.1 Hardware

It is by now commonplace to remark that core counts, both on a single chip and in a complete system, are increasing, with a corresponding increase in the complexity of the memory system – non-uniform memory access and multiple levels of cache sharing. Systems software, and in particular the OS, must tackle the complex problem of scheduling both OS tasks and those of applications across a number of processors based on memory locality.

At the same time, cores themselves are becoming non-uniform: Asymmetric multicore processors (AMP) [31] mix cores of different microarchitectures (and therefore performance and energy characteristics) on a single processor. A key motivation for this is power reduction for embedded systems like smartphones: under high CPU load, complex, high-performance cores can complete tasks more quickly, resulting in power reduction in other areas of the system. Under light CPU load, however, it is more efficient to run tasks on simple, low-power cores.

While migration between cores can be transparent to the OS (as is possible with, e.g., ARM's "big.LITTLE" AMP architecture) a better solution is for the OS to manage a heterogeneous collection of cores itself, powering individual cores on and off reactively.

Alternatively, Intel's Turbo Boost feature, which increases the frequency and voltage of a core when others on the same die are sufficiently idle to keep the chip within its thermal envelope, is arguably a dynamic form of AMP [15].

At the same time, *hotplug* of processors, once the province of specialized machines like the Tandem NonStop systems [6], is becoming more mainstream. More radical proposals for reconfiguring physical processors include Core Fusion [26], whereby multiple independent cores can be morphed into a larger CPU, pooling caches and functional units to improve the performance of sequential programs.

Ultimately, the age of "dark silicon" [21] may well lead to increased core counts, but with a hard limit on the number that may be powered on at any given time. Performance advances and energy savings subsequently will have to derive from specialized hardware for particular workloads or operations [47].

The implications for a future OS are that it must manage a dynamic set of physical cores, and be able to adjust to changes in the number, configuration, and microarchitecture of cores available at runtime, while maintaining a stable execution environment for applications.

2.2 Software

Alongside hardware trends, there is increasing interest in modifying, upgrading, patching, or replacing OS kernels at runtime. Baumann et al. [9] implement dynamic kernel updates in K42, leveraging the object-oriented design of the OS, and later extend this to interface changes using object adapters and lazy update [7]. More recently, Ksplice [3] allows binary patching of Linux kernels without reboot, and works by comparing generated object code and replacing entire functions. Dynamic instrumentation

[1] Barrelfish uses the term *CPU driver* to refer to the kernel-mode code running on a core. In this paper, we use the term "kernel" instead, to avoid confusion with *boot driver*.

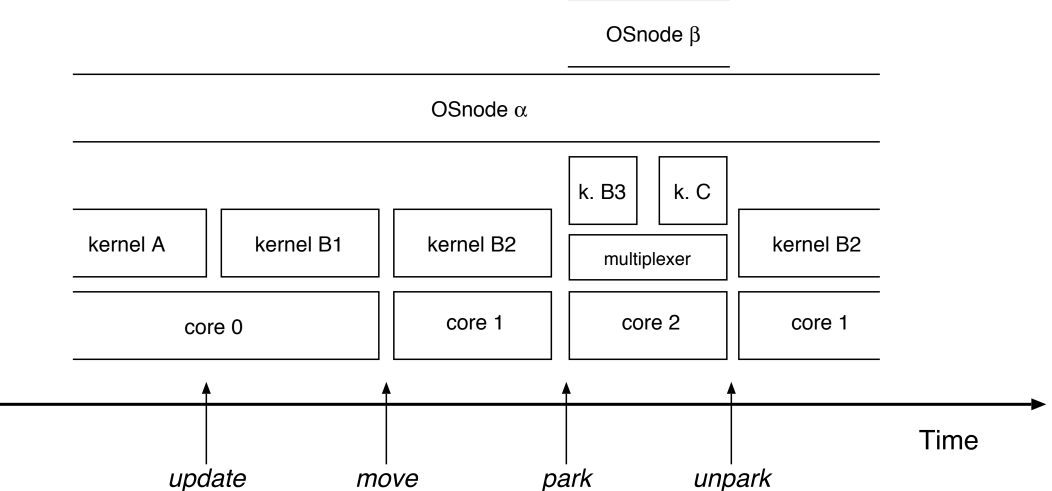


Figure 1: Shows the supported operations of a decoupled OS. **Update**: The entire kernel, dispatching OSnode α, is replaced at runtime. **Move**: OSnode α containing all per-core state, entailing applications is migrated to another core and kernel. **Park**: OSnode α is moved to a new core and kernel that temporarily dispatches two OSnodes. **Unpark**: OSnode α is transferred back to its previous core.

systems like Dtrace [13] provide mechanisms that modify the kernel at run-time to analyze program behavior.

All these systems show that the key challenges in updating an OS online are to maintain critical invariants across the update and to do so with minimal interruption of service (the system should pause, if at all, for a minimal period). This is particularly hard in a multiprocessor kernel with shared state.

In this paper, we argue for addressing *all* these challenges in a single framework for core and kernel management in the OS, although the structure of Unix-like operating systems presents a barrier to such a unified framework. The rest of this paper describes the unified approach we adopted in Barrelfish/DC.

3 Related work

Our work combines several directions in OS design and implementation: core hotplugging, kernel update and replacement, and multikernel architectures.

3.1 CPU Hotplug

Most modern OS designs today support some form of core hotplug. Since the overriding motivation is reliability, unplugging or plugging a core is considered a rare event and the OS optimizes the common case where the cores are not being hotplugged. For example, Linux CPU hotplug uses the `__stop_machine()` kernel call, which halts application execution on all online CPUs for typically hundreds of milliseconds [23], overhead that increases further when the system is under CPU load [25]. We show further evidence of this cost in Section 5.1 where we compare Linux' CPU hotplug with Barrelfish/DC' core update operations.

Recognizing that processors will be configured much more frequently in the future for reasons of energy usage and performance optimization, Chameleon [37] identifies several bottlenecks in the existing Linux implementation due to global locks, and argues that current OSes are ill equipped for processor sets that can be reconfigured at runtime. Chameleon extends Linux to provide support for changing the set of processors efficiently at runtime, and a scheduling framework for exploiting this new functionality. Chameleon can perform processor reconfiguration up to 100,000 times faster than Linux 2.6.

Barrelfish/DC is inspired in part by this work, but adopts a very different approach. Where Chameleon targets a single, monolithic shared kernel, Barrelfish/DC adopts a multikernel model and uses the ability to reboot individual kernels one by one to support CPU reconfiguration.

The abstractions provided are accordingly different: Chameleon abstracts hardware processors behind *processor proxies* and *execution objects*, in part to handle the problem of per-core state (primarily interrupt handlers) on an offline or de-configured processor. In contrast, Barrelfish/DC abstracts the per-core state (typically much larger in a shared-nothing multikernel than in a shared-memory monolithic kernel) behind OSnode and *kernel control block* abstractions.

In a very different approach, Kozuch et al. [30] show how commodity OS hibernation and hotplug facilities can be used to migrate a complete OS between different machines (with different hardware configurations) without virtualization.

Hypervisors are typically capable of simulating hotplugging of CPUs within a virtual machine. Barrelfish/DC can be deployed as a guest OS to manage a variable set of virtual CPUs allocated by the hypervisor. Indeed, Barrelfish/DC addresses a long-standing issue in virtualization: it is hard to fully virtualize the *microarchitecture* of a processor when VMs might migrate between asymmetric cores or between physical machines with different processors. As a guest, Barrelfish/DC can natively handle such heterogeneity and change without disrupting operation.

3.2 Kernel updates

The problem of patching system software without downtime of critical services has been a research area for some time. For example, K42 explored update of a running kernel [7, 9], exploiting the system's heavily object-oriented design. Most modern mainstream OSes support dynamic loading and unloading of kernel modules, which can be used to update or specialize limited parts of the OS.

KSplice [3] patches running Linux kernels without the need for reboot by replacing code in the kernel at a granularity of complete functions. It uses the Linux `stop_machine()` call to ensure that no CPU is currently executing a function to be replaced, and places a branch instruction at the start of the obsolete function to direct execution of the replacement code. Systems like KSplice replace individual functions across all cores at the same time. In contrast, Barrelfish/DC replaces entire kernels, but on a subset of cores at a time. KSplice makes sense for an OS where all cores must execute in the same, shared-memory kernel and the overhead incurred by quiescing the entire machine is unavoidable.

Proteos [22] uses a similar approach to Barrelfish/DC by replacing applications in their entirety instead of applying patches to existing code. In contrast to Ksplice, Proteos automatically applies state updates while preserving pointer integrity in many cases, which eases the burden on programmers to write complicated state transformation functions. In contrast to Barrelfish/DC, Proteos does not upgrade kernel-mode code but focuses on updates for OS processes running in user-space, in a micro-kernel environment. Much of the OS functionality in Barrelfish/DC resides in user-space as well, and Proteos would be applicable here.

Otherworld [18] also enables kernel updates without disrupting applications, with a focus on recovering system crashes. Otherworld can microreboot the system kernel after a critical error without clobbering running applications' state, and then attempt to restore applications that were running at the time of a crash by recreating application memory spaces, open files and other resources.

Rather than relying on a single, system-wide kernel, Barrelfish/DC exploits the multikernel environment to offer both greater flexibility and better performance: kernels and cores can be updated dynamically with (as we show in Section 5) negligible disruption to the rest of the OS.

While their goals of security and availability differ somewhat from Barrelfish/DC, KeyKOS [24] and EROS [42] use partitioned capabilities to provide an essentially stateless kernel. Memory in KeyKOS is persistent, and it allows updates of the OS while running, achieving continuity by restoring from disk-based checkpoints of the entire capability state. Barrelfish/DC by contrast achieves continuity by distributing the capability system, only restarting some of the kernels at a time, and preserving each kernel's portion of the capability system across the restart.

3.3 Multikernels

Multikernels such as fos [48], Akaros [40], Tessellation [33], Hive [14], and Barrelfish [8], are based on the observation that modern hardware is a networked system and so it is advantageous to model the OS as a distributed system. For example, Barrelfish runs a small kernel on each core in the system, and the OS is built as a set of cooperating processes, each running on one of these kernels, sharing no memory, and communicating via message passing. Multikernels are motivated by both the scalability advantages of sharing no cache lines between cores, and the goal of supporting future hardware with heterogeneous processors and little or no cache-coherent or shared physical memory.

Barrelfish/DC exploits the multikernel design for a new reason: dynamic and flexible management of the cores and the kernels of the system. A multikernel can naturally run different versions of kernels on different cores. These versions can be tailored to the hardware, or specialized for different workloads.

Furthermore, since (unlike in monolithic kernels) the state on each core is relatively decoupled from the rest of the system, multikernels are a good match for systems where cores come and go, and intuitively should support reconfiguration of part of the hardware without undue disruption to software running elsewhere on the machine. Finally, the shared-nothing multikernel architecture allows us to wrap kernel state and move it between different kernels without worrying about potentially harmful concurrent accesses.

We chose to base Barrelfish/DC on Barrelfish, as it is readily available, is under active development, supports multiple hardware platforms, and can run a variety of

common applications such as databases and web servers. The features of Barrelfish/DC described in this paper will be incorporated into a future Barrelfish release.

Recently, multikernels have been combined with traditional OS designs such as Linux [27, 36] so as to run multiple Linux kernels on different cores of the same machine using different partitions of physical memory, in order to provide performance isolation between applications. Popcorn Linux [38, 43] boots a modified Linux kernel in this fashion, and supports kernel- and user-space communication channels between kernels [41], and process migration between kernels. In principle, Popcorn extended with the ideas in Barrelfish/DC could be combined with Chameleon in a two-level approach to dynamic processor support.

4 Design

We now describe how Barrelfish/DC decouples cores, kernels, and the rest of the OS. We focus entirely on mechanism in this paper, and so do not address scheduling and policies for kernel replacement, core power management, or application migration. Note also that our main motivation in Barrelfish/DC is adapting the OS for performance and flexibility, and so we do not consider fault tolerance and isolation for now.

We first describe how Barrelfish/DC boots a new core, and then present in stages the problem of per-core state when removing a core, introducing the Barrelfish/DC capability system and kernel control block. We then discuss the challenges of time and interrupts, and finish with a discussion of the wider implications of the design.

4.1 Booting a new core

Current CPU hotplug approaches assume a single, shared kernel and a homogeneous (albeit NUMA) machine, with a variable number of active cores up to a fixed limit, and so a static in-kernel table of cores (whether active or inactive) suffices to represent the current hardware state. Bringing a core online is a question of turning it on, updating this table, and creating per-core state when needed. Previous versions of Barrelfish also adopted this approach, and booted all cores during system initialization, though there has been experimental work on dynamic booting of heterogeneous cores [35].

Barrelfish/DC targets a broader hardware landscape, with complex machines comprising potentially heterogeneous cores. Furthermore, since Barrelfish/DC runs a different kernel instance on each core, there is no reason why the same kernel code should run everywhere – indeed, we show one advantage of *not* doing this in Section 5.3. We thus need an OS representation of a core on the machine which abstracts the hardware-dependent mechanisms for bringing that core up (with some kernel) and down.

Therefore, Barrelfish/DC introduces the concept of a *boot driver*, which is a piece of code running on a "home core" which manages a "target core" and encapsulates the hardware functionality to boot, suspend, resume, and power-down the latter. Currently boot drivers run as processes, but closely resemble device drivers and could equally run as software objects within another process.

A new core is brought online as follows:

1. The new core is detected by some platform-specific mechanism (e.g., ACPI) and its appearance registered with the device management subsystem.
2. Barrelfish/DC selects and starts an appropriate boot driver for the new core.
3. Barrelfish/DC selects a kernel binary and arguments for the new core, and directs the boot driver to boot the kernel on the core.
4. The boot driver loads and relocates the kernel, and executes the hardware protocol to start the new core.
5. The new kernel initializes and uses existing Barrelfish protocols for integrating into the running OS.

The boot driver abstraction treats CPU cores much like peripheral devices, and allows us to reuse the OS's existing device and hotplug management infrastructure [50] to handle new cores and select drivers and kernels for them. It also separates the hardware-specific *mechanism* for booting a core from the *policy* question of what kernel binary to boot the core with.

Boot drivers remove most of the core boot process from the kernel: in Barrelfish/DC we have entirely replaced the existing multiprocessor booting code for multiple architectures (which was spread throughout the system) with boot drivers, resulting in a much simpler system structure, and reduced code in the kernels themselves.

Booting a core (and, indeed, shutting it down) in Barrelfish/DC only involves two processes: the boot driver on the home core, and the kernel on the target core. For this reason, we require no global locks or other synchronization in the system, and the performance of these operations is not impacted by load on other cores. We demonstrate these benefits experimentally in Section 5.1.

Since a boot driver for a core requires (as with a device driver) at least one existing core to execute, there is a potential dependency problem as cores come and go. For the PC platform we focus on here, this is straightforward since any core can run a boot driver for any other core, but we note that in general the problem is the same as that of allocating device drivers to cores.

Boot drivers provide a convenient abstraction of hardware and are also used to shutdown cores, but this is *not* the main challenge in removing a core from the system.

4.2 Per-core state

Taking a core out of service in a modern OS is a more involved process than booting it, since modern multicore OSes include varying amounts of per-core kernel state. If they did not, removing a core would be simply require migrating any running thread somewhere else, updating the scheduler, and halting the core.

The challenge is best understood by drawing a distinction between the *global* state in an OS kernel (i.e., the state which is shared between all running cores in the system) and the *per-core* state, which is only accessed by a single core. The kernel state of any OS is composed of these two categories.

In, for example, older versions of Unix, all kernel state was global and protected by locks. In practice, however, a modern OS keeps per-core state for scalability of scheduling, memory allocation, virtual memory, etc. Per-core data structures reduce write sharing of cache lines, which in turn reduces interconnect traffic and cache miss rate due to coherency misses.

For example, Linux and Windows use per-core scheduling queues, and distributed memory allocators. Corey [10] allowed configurable sharing of page tables between cores, and many Linux scaling enhancements (e.g., [11]) have been of this form. K42 [2] adopted reduced sharing as a central design principle, and introduced the abstraction of *clustered objects*, essentially global proxies for pervasive per-core state.

Multikernels like Barrelfish [8] push this idea to its logical conclusion, sharing no data (other than message channels) between cores. Multikernels are an extreme point in the design space, but are useful for precisely this reason: they highlight the problem of consistent per-core state in modern hardware. As core counts increase, we can expect the percentage of OS state that is distributed in more conventional OSes to increase.

Shutting down a core therefore entails disposing of this state without losing information or violating system-wide consistency invariants. This may impose significant overhead. For example, Chameleon [37] devotes considerable effort to ensuring that per-core interrupt handling state is consistent across CPU reconfiguration. As more state becomes distributed, this overhead will increase.

Worse, how to dispose of this state depends on what it is: removing a per-core scheduling queue means migrating threads to other cores, whereas removing a per-core memory allocator requires merging its memory pool with another allocator elsewhere.

Rather than implementing a succession of piecemeal solutions to this problem, in Barrelfish/DC we adopt a radical approach of lifting *all* the per-core OS state out of the kernel, so that it can be reclaimed lazily without delaying the rest of the OS. This design provides the means to completely decouple per-core state from both the underlying kernel implementation and the core hardware.

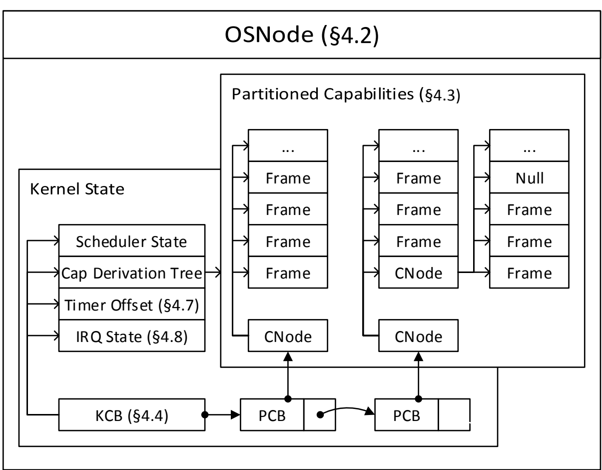


Figure 2: State in the Barrelfish/DC OSnode

We find it helpful to use the term *OSnode* to denote the total state of an OS kernel local to a particular core. In Linux the OSnode changes with different versions of the kernel; Chameleon identifies this state by manual annotation of the kernel source code. In Barrelfish, the OSnode is all the state – there is no shared global data.

4.3 Capabilities in Barrelfish/DC

Barrelfish/DC captures the OSnode using its capability system: all memory and other resources maintained by the core (including interrupts and communication end-points) are represented by capabilities, and thus the OSnode is represented by the capability set of the core. The per-core state of Barrelfish/DC is shown schematically in Figure 2.

Barrelfish/DC's capability system, an extension of that in Barrelfish [44], is derived from the *partitioned capability* scheme used in seL4 [19, 20, 28].

In seL4 (and Barrelfish), all regions of memory are referred to by capabilities, and capabilities are *typed* to reflect what the memory is used for. For example, a "frame" capability refers to memory that the holder can map into their address space, while a "c-node" capability refers to memory that is used to store the bit representations of capabilities themselves. The security of the system as a whole derives from the fact that only a small, trusted computing base (the kernel) holds both a frame capability and a c-node capability to the same memory, and can therefore fabricate capabilities.

A capability for a region can be split into two smaller regions, and also *retyped* according to a set of system rules that preserve integrity. Initially, memory regions are of type "untyped", and must be explicitly retyped to

"frame", "c-node", or some other type.

This approach has the useful property that a process can allocate memory without being able to access its contents. This is used in seL4 to remove *any* dynamic memory allocation from the kernel, greatly simplifying both the formal specification of the kernel and its subsequent proof [20]. All kernel objects (such as process control blocks, or page tables) are allocated by user-level processes which can, themselves, not access them directly.

A key insight of Barrelfish/DC is that this approach can externalize the kernel state *entirely,* as follows.

4.4 Kernel Control Blocks

In developing Barrelfish/DC, we examined the Barrelfish kernel to identify all the data structures which were not direct (optimized) derivations of information already held in the capability tree (and which could therefore be reconstructed dynamically from the tree). We then eliminated from this set any state that did not need to persist across a kernel restart.

For example, the runnable state and other scheduling parameters of a process[2] are held in the process' control block, which is part of the capability system. However, the scheduler queues themselves do not need to persist across a change of kernel, since (a) any scheduler will need to recalculate them based on the current time, and (b) the new scheduler may have a completely different policy and associated data structures anyway.

What remained was remarkably small: it consists of:

- The minimal scheduling state: the head of a linked list of a list of process control blocks.
- Interrupt state. We discuss interrupts in Section 4.8.
- The root of the capability derivation tree, from which all the per-core capabilities can be reached.
- The timer offset, discussed in Section 4.7.

In Barrelfish/DC, we introduce a new memory object, the *Kernel Control Block* (KCB), and associated capability type, holding this data in a standard format. The KCB is small: for 64-bit x86 it is about 28 KiB in size, almost all of which is used by communication endpoints for interrupts.

4.5 Replacing a kernel

The KCB effectively decouples the per-core OS state from the kernel. This allows Barrelfish/DC to shut down a kernel on a core (under the control of the boot driver running on another core) and replace it with a new one. The currently running kernel saves a small amount of persistent state in the KCB, and halts the core. The boot driver then loads a new kernel with an argument supplying the address of the KCB. It then restarts the core (using an IPI on x86 machines), causing the new kernel to boot. This new kernel then initializes any internal data structures it needs from the KCB and the OSnode capability database.

The described technique allows for arbitrary updates of kernel-mode code. By design, the kernel does not access state in the OSnode concurrently. Therefore, having a quiescent state in the OSnode before we shut-down a core is always guaranteed. The simplest case for updates requires no changes in any data structures reachable by the KCB and can be performed as described by simply replacing the kernel code. Updates that require a transformation of the data structures may require a one-time adaption function to execute during initialization, whose overhead depends on the complexity of the function and the size of the OSnode. The worst-case scenario is one that requires additional memory, since the kernel by design delegates dynamic memory allocation to userspace.

As we show in Section 5, replacing a kernel can be done with little performance impact on processes running on the core, even device drivers.

4.6 Kernel sharing and core shutdown

As we mentioned above, taking a core completely out of service involves not simply shutting down the kernel, but also disposing of or migrating all the per-core state on the core, and this can take time. Like Chameleon, Barrelfish/DC addresses this problem by deferring it: we immediately take the core down, but keep the OSnode running in order to be able to dismantle it lazily. To facilitate this, we created a new kernel which is capable of multiplexing several KCBs (using a simple extension to the existing scheduler).

Performance of two active OSnodes sharing a core is strictly best-effort, and is not intended to be used for any case where application performance matters. Rather, it provides a way for an OSnode to be taken out of service in the background, after the core has been shut down.

Note that there is no need for all cores in Barrelfish/DC to run this multiplexing kernel, or, indeed, for any cores to run it when it is not being used – it can simply replace an existing kernel on demand. In practice, we find that there is no performance loss when running a single KCB above a multiplexing kernel.

Decoupling kernel state allows attaching and detaching KCBs from a running kernel. The entry point for kernel code takes a KCB as an argument. When a new kernel is started, a fresh KCB is provided to the kernel code. To restart a kernel, the KCB is detached from the running kernel code, the core is shut down, and the KCB is provided to the newly booted kernel code.

[2]Technically, it is a Barrelfish "dispatcher", the core-local representation of a process. A process usually consists of a set of distinct "dispatchers", one in each OSnode.

We rely on shared physical memory when moving OSnodes between cores. This goes against the original multikernel premise that assumes no shared memory between cores. However, an OSnode is still always in use by strictly one core at the time. Therefore, the benefits of avoiding concurrent access in OSnode state remain. We discuss support for distributed memory hardware in Section 6.

The combination of state externalization via the KCB and kernel sharing on a single core has a number of further applications, which we describe in Section 4.10.

4.7 Dealing with time

One of the complicating factors in starting the OSnode with a new kernel is the passage of time. Each kernel maintains a per-core internal clock (based on a free-running timer, such as the local APIC), and expects this to increase monotonically. The clock is used for per-core scheduling and other time-sensitive tasks, and is also available to application threads running on the core via a system call.

Unfortunately, the hardware timers used are rarely synchronized between cores. Some hardware (for example, modern PCs) define these timers to run at the same rate on every core (regardless of power management), but they may still be offset from each other. On other hardware platforms, these clocks may simply run at different rates between cores.

In Barrelfish/DC we address this problem with two fields in the KCB. The first holds a constant offset from the local hardware clock; the OS applies this offset whenever the current time value is read.

The second field is set to the current local time when the kernel is shut down. When a new kernel starts with an existing KCB, the offset field is reinitialized to the difference between this old time value and the current hardware clock, ensuring that local time for the OSnode proceeds monotonically.

4.8 Dealing with interrupts

Interrupts pose an additional challenge when moving an OSnode between cores. It is important that interrupts from hardware devices are always routed to the correct kernel. In Barrelfish interrupts are then mapped to messages delivered to processes running on the target core. Some interrupts (such as those from network cards) should "follow" the OSnode to its new core, whereas others should not. We identify three categories of interrupt.

1. Interrupts which are used exclusively by the kernel, for example a local timer interrupt used to implement preemptive scheduling. Handling these interrupts is internal to the kernel, and their sources are typically per-core hardware devices like APICs or performance counters. In this case, there is no need to take additional actions when reassigning KCBs between cores.
2. Inter-processor interrupts (IPIs), typically used for asynchronous communication between cores. Barrelfish/DC uses an indirection table that maps OSnode identifiers to the physical core running the corresponding kernel. When one kernel sends an IPI to another, it uses this table to obtain the hardware destination address for the interrupt. When detaching a KCB from a core, its entry is updated to indicate that its kernel is unavailable. Similarly, attaching a KCB to a core, updates the location to the new core identifier.
3. Device interrupts, which should be forwarded to a specific core (e.g. via IOAPICs and PCIe bridges) running the handler for the device's driver.

When Barrelfish/DC device drivers start up they request forwarding of device interrupts by providing two capability arguments to their local kernel: an opaque interrupt descriptor (which conveys authorization to receive the interrupt) and a message binding. The interrupt descriptor contains all the architecture-specific information about the interrupt source needed to route the interrupt to the right core. The kernel associates the message binding with the architectural interrupt and subsequently forwards interrupts to the message channel.

For the device and the driver to continue normal operation, the interrupt needs to be re-routed to the new core, and a new mapping is set up for the (existing) driver process. This could be done either transparently by the kernel, or explicitly by the device driver.

We choose the latter approach to simplify the kernel. When a Barrelfish/DC kernel shuts down, it disables all interrupts. When a new kernel subsequently resumes an OSnode, it sends a message (via a scheduler upcall) to every process which had an interrupt registered. Each driver process responds to this message by re-registering its interrupt, and then checking with the device directly to see if any events have been missed in the meantime (ensuring any race condition is benign). In Section 5.2.1 we show the overhead of this process.

4.9 Application support

From the perspective of applications which are oblivious to the allocation of physical cores (and which deal solely with threads), the additional functionality of Barrelfish/DC is completely transparent. However, many applications such as language runtimes and database systems deal directly with physical cores, and tailor their scheduling of user-level threads accordingly.

For these applications, Barrelfish/DC can use the existing scheduler activation [1] mechanism for process dispatch in Barrelfish to notify userspace of changes in the number of online processors, much as it can already convey the allocation of physical cores to applications.

4.10 Discussion

From a broad perspective, the combination of boot drivers and replaceable kernels is a radically different view of how an OS should manage processors on a machine. Modern general-purpose kernels such as Linux try to support a broad set of requirements by implementing different behaviors based on build-time and run-time configuration. Barrelfish/DC offers an alternative: instead of building complicated kernels that try to do many things, build simple kernels that do one thing well. While Linux selects a single kernel at boot time for all cores, Barrelfish/DC allows selecting not only per-core kernels, but changing this selection on-the-fly.

There are many applications for specialized kernels, including those tailored for running databases or language run-times, debugging or profiling, or directly executing verified user code as in Google's native client [49].

To take one example, in this paper we demonstrate support for hard real-time applications. Despite years of development of real-time support features in Linux and other general-purpose kernels [16], many users resort to specialized real-time OSes, or modified versions of general-purpose OSes [32].

Barrelfish/DC can offer hard real-time support by rebooting a core with a specialized kernel, which, to eliminate OS jitter, has no scheduler (since it targets a single application) and takes no interrupts. If a core is not preallocated, it must be made available at run-time by migrating the resident OSnode to another core that runs a multi-KCB kernel, an operation we call *parking*. If required, cache interference from other cores can also be mitigated by migrating *their* OSnodes to other packages. Once the hard real-time application finishes, the OSnodes can be moved back to the now-available cores. We evaluate this approach in Section 5.3.

5 Evaluation

We present here a performance evaluation of Barrelfish/DC. First (Section 5.1), we measure the performance of starting and stopping cores in Barrelfish/DC and in Linux. Second (Section 5.2), we investigate the behavior of applications when we restart kernels, and when we park OSnodes. Finally, (Section 5.3), we demonstrate isolating performance via a specialized kernel. We perform experiments on the set of x86 machines shown in Table 1. Hyperthreading, TurboBoost, and SpeedStep technologies are disabled in machines that support them, as they complicate cycle counter measurements. TurboBoost and SpeedStep can change the processor frequency in unpredictable ways, leading to high fluctuation for repeated experiments. The same is true for Hyperthreading due to sharing of hardware logic between logical cores. However, TurboBoost and Hyperthreading are both relevant for this work as discussed in Section 6 and Section 1.

packages×cores/uarch	CPU model
2×2 Santa-Rosa	2.8 GHz Opteron 2200
4×4 Shanghai	2.5 GHz Opteron 8380
2×10 SandyBridge	2.5 GHz Xeon E5-2670 v2
1×4 Haswell	3.4 GHz Xeon E3-1245 v3

Table 1: Systems we use in our evaluation. The first column describes the topology of the machine (total number of packages and cores per package) and the second the CPU model.

5.1 Core management operations

In this section, we evaluate the performance of managing cores in Barrelfish/DC, and also in Linux using the CPU Hotplug facility [4]. We consider two operations: shutting down a core (*down*) and bringing it back up again (*up*).

Bringing up a core in Linux is different from bringing up a core in Barrelfish/DC. In Barrelfish/DC, each core executes a different kernel which needs to be loaded by the boot driver, while in Linux all cores share the same code. Furthermore, because cores share state in Linux, core management operations require global synchronization, resulting in stopping application execution in all cores for an extended period of time [23]. Stopping cores is also different between Linux and Barrelfish/DC. In Linux, applications executed in the halting core need to be migrated to other online cores before the shutdown can proceed, while in Barrelfish/DC we typically would move a complete OSnode after the shutdown and not individual applications.

In Barrelfish/DC, the *down* time is the time it takes the boot driver to send an appropriate IPI to the core to be halted plus the propagation time of the IPI and the cost of the IPI handler in the receiving core. For the *up* operation we take two measurements: the boot driver cost to prepare a new kernel up until (and including) the point where it sends an IPI to the starting core (*driver*), and the cost in the booted core from the point it wakes up until the kernel is fully online (*core*).

In Linux, we measure the latency of starting or stopping a core using the log entry of the `smpboot` module and a sentinel line echoed to `/dev/kmsg`. For core shutdown, `smboot` reports when the core becomes offline, and we insert the sentinel right before the operation is initiated.

	Barrelfish/DC						Linux			
	idle			load			idle		load	
	down (µs)	up driver (ms)	up core (ms)	down (µs)	up driver (ms)	up core (ms)	down (ms)	up (ms)	down (ms)	up (ms)
2×2 Santa-Rosa	2.7 / —[a]	29	1.2	2.7 / —	34 ± 17	1.2	131 ± 25	20 ± 1	5049 ± 2052	26 ± 5
4×4 Shanghai	2.3 / 2.6	24	1.0	2.3 / 2.7	46 ± 76	1.0	104 ± 50	18 ± 3	3268 ± 980	18 ± 3
2×10 SandyBridge	3.5 / 3.7	10	0.8	3.6 / 3.7	23 ± 52	0.8	62 ± 46	21 ± 7	2265 ± 1656	23 ± 5
1×4 Haswell	0.8 / —[a]	7	0.5	0.8 / —	7 ± 0.1	0.5	46 ± 40	14 ± 1	2543 ± 1710	20 ± 5
	Results in cycles									
	$\times 10^3$	$\times 10^6$	$\times 10^6$	$\times 10^3$	$\times 10^6$	$\times 10^6$	$\times 10^6$	$\times 10^6$	$\times 10^6$	$\times 10^6$
2×2 Santa-Rosa	8 / —	85	3.4	8 / —	97 ± 49	3.5	367 ± 41	56 ± 2.0	14139 ± 5700	74 ± 21
4×4 Shanghai	6 / 6	63	2.6	6 / 7	115 ± 192	2.6	261 ± 127	44 ± 2.0	8170 ± 2452	46 ± 8
2×10 SandyBridge	9 / 10	27	2.1	9 / 10	59 ± 133	2.1	155 ± 116	53 ± 2.0	5663 ± 4141	57 ± 12
1×4 Haswell	3 / —	26	1.9	2.9 / —	26 ± 0.40	2.0	156 ± 137	50 ± 0.5	8647 ± 5816	69 ± 16

Table 2: Performance of core management operations for Barrelfish/DC and Linux (3.13) when the system is idle and when the system is under load. For the Barrelfish/DC *down* column, the value after the slash shows the cost of stopping a core on another socket with regard to the boot driver. [a]We do not include this number for Santa-Rosa because it lacks synchronized timestamp counters, nor for Haswell because it only includes a single package.

For core boot, `smpboot` reports when the operation starts, so we insert the sentinel line right after the operation.

For both Barrelfish/DC and Linux we consider two cases: an idle system (*idle*), and a system with all cores under load (*load*). In Linux, we use the `stress` tool [45] to spawn a number of workers equal to the number of cores that continuously execute the `sync` system call. In Barrelfish/DC, since the file-system is implemented as a user-space service, we spawn an application that continuously performs memory management system calls on each core of the system.

Table 2 summarizes our results. We show both time (msecs and µsecs) and cycle counter units for convenience. All results are obtained by repeating the experiment 20 times, and calculating the mean value. We include the standard deviation where it is non-negligible.

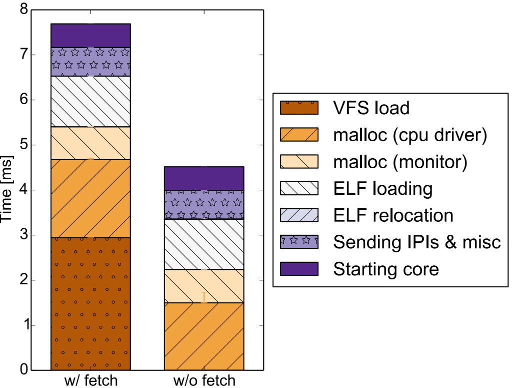


Figure 3: Breakdown of the cost of bringing up a core for the Haswell machine.

Stopping cores: The cost of stopping cores in Barrelfish/DC ranges from 0.8 µs (Haswell) to 3.5 µs (SandyBridge). Barrelfish/DC does not share state across cores, and as a result no synchronization between cores is needed to shut one down. Furthermore, Barrelfish/DC' shutdown operation consists of sending an IPI, which will cause the core to stop after a minimal operation in the KCB (saving the timer offset). In fact, the cost of stopping a core in Barrelfish/DC is small enough to observe the increased cost of sending an IPI across sockets, leading to an increase of 5% in stopping time on SandyBridge and 11% on Shanghai. These numbers are shown in Table 2, in the Barrelfish/DC *down* columns after the slash. As these measurements rely on timestamp counters being synchronized across packages, we are unable to present the cost increase of a cross-socket IPI on the Santa-Rosa machine whose timestamp counters are only synchronized within a single package.

In stark contrast, the cost of shutting down a core in Linux ranges from 46 ms to 131 ms. More importantly, the shutdown cost in Linux explodes when applying load, while it generally remains the same for Barrelfish/DC. For example, the average time to power down a core in Linux on Haswell is increased by 55 times when we apply load.

Starting cores: For Barrelfish/DC, the setup cost in the boot driver (*driver*) dominates the cost of starting a core (*core*). Fig. 3 shows a breakdown of the costs for bringing up a core on Haswell. *Starting core* corresponds to the

core Table 2 column, while the rest corresponds to operations performed by the boot driver: loading the image from storage, allocating memory, ELF loading and relocation, etc. Loading the kernel from the file system is the most expensive operation. If multiple cores are booted with the same kernel, this image can be cached, significantly improving the time to start a core as shown in the second bar in Fig. 3. We note that the same costs will dominate the restart operation since shutting down a core has negligible cost compared to bringing it up. Downtime can be minimized by first doing the necessary preparations in the boot driver and then halting and starting the core.

Even though Barrelfish/DC has to prepare the kernel image, when idle, the cost of bringing up a core for Barrelfish/DC is similar to the Linux cost (Barrelfish/DC is faster on our Intel machines, while the opposite is true for our AMD machines). Bringing a core up can take from 7 ms (Barrelfish/DC/Haswell) to 29 ms (Barrelfish/DC/Santa-Rosa). Load affects the cost of booting up a core to varying degrees. In Linux such an effect is not observed in the Shanghai machine, while in the Haswell machine load increases average start time by 33%. The effect of load when starting cores is generally stronger in Barrelfish/DC (e.g., in SandyBridge the cost is more than doubled) because the boot driver time-shares its core with the load generator.

Overall, Barrelfish/DC has minimal overhead stopping cores. For starting cores, results vary significantly across different machines but the cost of bringing up cores in Barrelfish/DC is comparable to the respective Linux cost.

5.2 Applications

In this section, we evaluate the behavior of real applications under two core management operations: *restarting*, where we update the core kernel as the application runs, and *parking*. In parking, we run the application in a core with a normal kernel and then move its OSnode into a multi-KCB kernel running on a different core. While the application is parked it will share the core with another OSnode. We use a naive multi-KCB kernel that runs each KCB for 20 ms, which is two times the scheduler time slice. Finally, we move the application back to its original core. The application starts by running alone on its core. We execute all experiments in the Haswell machine.

5.2.1 Ethernet driver

Our first application is a Barrelfish NIC driver for the Intel 82574, which we modify for Barrelfish/DC to re-register its interrupts when instructed by the kernel (see Section 4.8). During the experiment we use `ping` from a client machine to send ICMP echo requests to the NIC. We run `ping` as root with the `-A` switch, where the inter-packet intervals adapt to the round-trip time. The `ping` manual states: "on networks with low rtt this mode is essentially equivalent to flood mode."

Fig. 4a shows the effect of the restart operation in the round-trip time latency experienced by the client. Initially, the ping latency is 0.042 ms on average with small variation. Restarting the kernel produces two outliers (packets 2307 and 2308 with an RTT of 11.1 ms and 1.07 ms, respectively). Note that 6.9 ms is the measured latency to bring up a core on this machine (Table 2).

We present latency results for the parking experiment in a timeline (Fig. 4b), and in a cumulative distribution function (CDF) graph (Fig. 4c). Measurements taken when the driver's OSnode runs exclusively on a core are denoted *Exclusive*, while measurements where the OSnode shares the core are denoted *Shared*. When parking begins, we observe an initial latency spike (from 0.042 ms to 73.4 ms). The spike is caused by the parking operation, which involves sending a KCB capability reference from the boot driver to the multi-KCB kernel as a message.[3] After the initial coordination, outliers are only caused by KCB time-sharing (maximum: 20 ms, mean: 5.57 ms). After unparking the driver, latency returns to its initial levels. Unparking does not cause the same spike as parking because we do not use messages: we halt the multi-KCB kernel and directly pass the KCB reference to a newly booted kernel.

5.2.2 Web server

In this experiment we examine how a web server[4] that serves files over the network behaves when its core is restarted and when its OSnode is parked. We initiate a transfer on a client machine in the server's LAN using `wget` and plot the achieved bandwidth for each 50 KiB chunk when fetching a 1 GiB file.

Fig. 4d shows the results for the kernel restart experiment. The effect in this case is negligible on the client side. We were unable to pinpoint the exact location of the update taking place from the data measured on the client and the actual download times during kernel updates were indistinguishable from a normal download. As expected, parking leads to a number of outliers caused by KCB time-sharing (Figures 4e and 4f). The average bandwidth before the parking is 113 MiB/s and the standard deviation 9 MiB/s, whereas during parking the average bandwidth is slightly lower at 111 MiB/s with a higher standard deviation of 19 MiB/s.

[3] We follow the Barrelfish approach, where kernel messages are handled by the *monitor*, a trusted OS component that runs in user-space.

[4] The Barrelfish native web server.

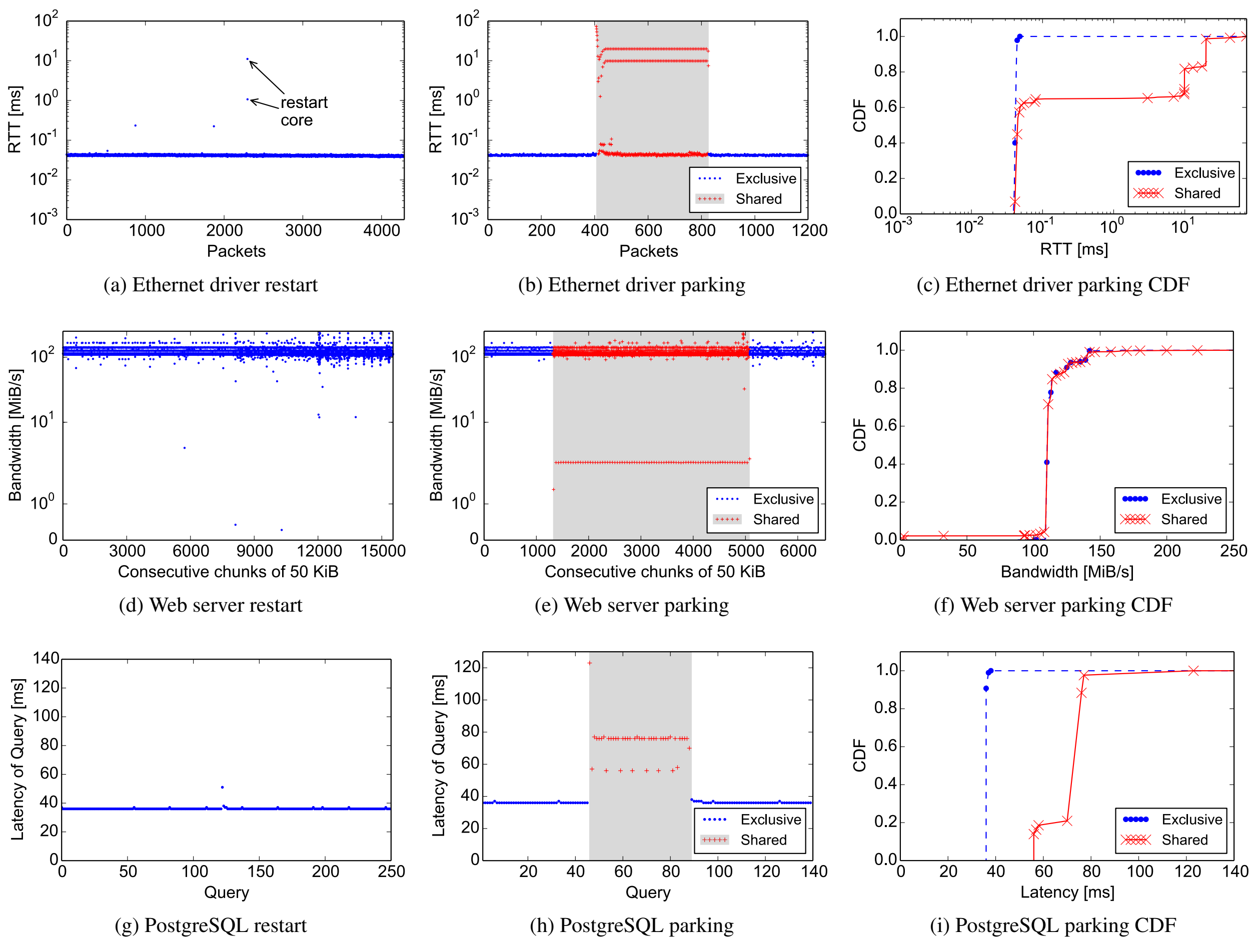


Figure 4: Application behavior when restarting kernels and parking OSnodes. For each application we include a timeline graph for restarting, and a timeline and a CDF graph for parking.

5.2.3 PostgreSQL

Next, we run a PostgreSQL [39] database server in Barrelfish/DC, using TPC-H [46] data with a scaling factor of 0.01, stored in an in-memory file-system. We measure the latency of a repeated CPU-bound query (query 9 in TPC-H) on a client over a LAN.

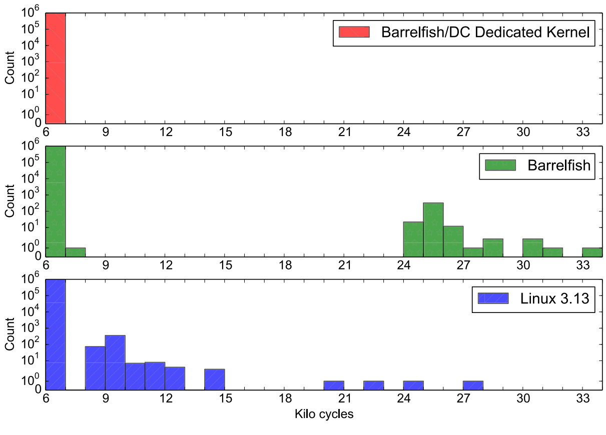
Fig. 4g shows how restart affects client latency. Before rebooting, average query latency is 36 ms. When a restart is performed, the first query has a latency of 51 ms. After a few perturbed queries, latency returns to its initial value.

Figures 4h and 4i show the effect of parking the OSnode that contains the PostgreSQL server. As before, during normal operation the average latency is 36 ms. When the kernel is parked we observe two sets of outliers: one (with more points) with a latency of about 76 ms, and one with latency close to 56 ms. This happens, because depending on the latency, some queries wait for two KCB time slices (20 ms each) of the co-hosted kernel, while others wait only for one.

Overall, we argue that kernel restart incurs acceptable overhead for online use. Parking, as expected, causes a performance degradation, especially for latency-critical applications. This is, however, inherent in any form of resource time-sharing. Furthermore, with improved KCB-scheduling algorithms the performance degradation can be reduced or tuned (e.g., via KCB priorities).

5.3 Performance isolation

Finally, we illustrate the benefits of Barrelfish/DC' support for restarting cores with specialized kernels using the case of hard-real time applications where eliminating OS jitter is required. To ensure that the application will run uninterrupted, we assign a core with a specialized kernel that does not implement scheduling and does not handle interrupts (see Section 4.10). We evaluate the performance isolation that can be achieved with our



(a) Histogram for all samples

(b) CDF for samples in the range of 6–7k cycles

Figure 5: Number of cycles measured for 10^3 iterations of a synthetic benchmark for Barrelfish/DC, Barrelfish, and Linux using real-time priorities.

specialized kernel compared to the isolation provided by: (i) an unmodified Barrelfish kernel, and (ii) a Linux 3.13 kernel where we set the application to run with real-time priority. We run our experiments on the Haswell machine, ensuring that no other applications run on the same core.

To measure OS jitter we use a synthetic benchmark that only performs memory stores to a single location. Our benchmark is intentionally simple to minimize performance variance caused by architectural effects. We sample the timestamp counter every 10^3 iterations, for a total of 10^6 samples. Fig. 5a shows a histogram of sampled cycles, where for all systems, most of the values fall into the 6-7 thousand range (i.e., 6-7 cycles latency per iteration). Fig. 5b presents the CDF graph for the 6–7 kcycles range, showing that there are no significant differences for the three systems in this range.

Contrarily to the Barrelfish/DC dedicated kernel where *all* of the samples are in the 6-7k range, in Linux and Barrelfish we observe significant outliers that fall outside this range. Since we run the experiment on the same hardware, under the same configuration, we attribute the outliers to OS jitter. In Barrelfish the outliers reach up to 68k cycles (excluded from the graph). Linux performs better than Barrelfish, but its outliers still reach 27–28 kcycles. We ascribe the worse behavior of Barrelfish compared to Linux to OS services running in user-space.

We conclude that Barrelfish/DC enables the online deployment of a dedicated, simple to build, OS kernel that eliminates OS jitter and provides hard real-time guarantees.

6 Future directions

Our ongoing work on Barrelfish/DC includes both exploring the broader applications of the ideas, and also removing some of the existing limitations of the system.

On current hardware, we plan to investigate the power-management opportunities afforded by the ability to replace cores and migrate the running OS around the hardware. One short-term opportunity is to fully exploit Intel's Turbo Boost feature to accelerate a serial task by temporarily vacating (and thereby cooling) neighboring cores on the same package.

We also intend to use core replacement as a means to improve OS instrumentation and tracing facilities, by dynamically instrumenting kernels running on particular cores at runtime, removing all instrumentation overhead in the common case. Ultimately, as kernel developers we would like to minimize whole-system reboots as much as possible by replacing single kernels on the fly.

Barrelfish/DC currently assumes cache-coherent cores, where the OS state (i.e., the OSnode) can be easily migrated between cores by passing physical addresses. The lack of cache-coherency per se can be handled with suitable cache flushes, but on hardware platforms without shared memory, or with different physical address spaces on different cores, the OSnode might not require considerable transformation to migrate between cores. The Barrelfish/DC capability system *does* contain all the information necessary to correctly swizzle pointers when copying the OSnode between nodes, but the copy is likely to be expensive, and dealing with shared-memory application state (which Barrelfish fully supports outside the OS) is a significant challenge.

A somewhat simpler case to deal with is moving an OSnode between a virtual and physical machine, allowing the OS to switch from running natively to running in a VM container.

Note that there is no requirement for the boot driver to share memory with its target core, as long as it has a mechanism for loading a kernel binary into the latter's address space and controlling the core itself.

When replacing kernels, Barrelfish/DC assumes that the OSnode format (in particular, the capability system)

remains unchanged. If the in-memory format of the capability database changes, then the new kernel must perform a one-time format conversion when it boots. It is unclear how much of a limitation this is in practice, since the capability system of Barrelfish has changed relatively little since its inception, but one way to mitigate the burden of writing such a conversion function is to exploit the fact that the format is already specified in a domain-specific high-level language called Hamlet [17] to derive the conversion function automatically.

While Barrelfish/DC decouples cores, kernels, and the OS state, the topic of appropriate *policies* for using these mechanisms without user intervention is an important area for future work. We plan to investigate policies that, based on system feedback, create new kernels to replace others, and move OSnodes across cores.

Finally, while Barrelfish/DC applications are notified when the core set they are running on changes (via the scheduler activations mechanism), they are currently insulated from knowledge about hardware core reconfigurations. However, there is no reason why this must always be the case. There may be applications (such as databases, or language runtimes) which can benefit from being notified about such changes to the running system, and we see no reason to hide this information from applications which can exploit it.

7 Conclusion

Barrelfish/DC presents a radically different vision of how cores are exploited by an OS and the applications running above it, and implements it in a viable software stack: the notion that OS state, kernel code, and execution units should be decoupled and freely interchangeable. Barrelfish/DC is an OS whose design assumes that all cores are dynamic.

As hardware becomes more dynamic, and scalability concerns increase the need to partition or replicate state across cores, system software will have to change its assumptions about the underlying platform, and adapt to a new world with constantly shifting hardware. Barrelfish/DC offers one approach to meeting this challenge.

8 Acknowledgements

We would like to thank the anonymous reviews and our shepherd, Geoff Voelker, for their encouragement and helpful suggestions. We would also like to acknowledge the work of the rest of the Barrelfish team at ETH Zurich without which Barrelfish/DC would not be possible.

References

[1] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems 10*, 1 (1992), 53–79.

[2] APPAVOO, J., DA SILVA, D., KRIEGER, O., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., WATERLAND, A., WISNIEWSKI, R. W., XENIDIS, J., STUMM, M., AND SOARES, L. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems 25*, 3 (2007).

[3] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the EuroSys Conference* (2009), pp. 187–198.

[4] ASHOK RAJ. CPU hotplug support in the Linux kernel. `https://www.kernel.org/doc/Documentation/cpu-hotplug.txt`.

[5] The Barrelfish Operating System. `http://www.barrelfish.org/`, 12.04.14.

[6] BARTLETT, J. F. A NonStop Kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (1981), pp. 22–29.

[7] BAUMANN, A., APPAVOO, J., WISNIEWSKI, R. W., SILVA, D. D., KRIEGER, O., AND HEISER, G. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proceedings of the USENIX Annual Technical Conference* (2007), pp. 1–14.

[8] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (2009), pp. 29–44.

[9] BAUMANN, A., HEISER, G., APPAVOO, J., DA SILVA, D., KRIEGER, O., WISNIEWSKI, R. W., AND KERR, J. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference* (2005), pp. 279–291.

[10] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (2008), pp. 43–57.

[11] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), pp. 1–8.

[12] BUTLER, M., BARNES, L., SARMA, D. D., AND GELINAS, B. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro 31*, 2 (Mar. 2011), 6–15.

[13] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference* (2004), pp. 15–28.

[14] CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (1995), pp. 12–25.

[15] CHARLES, J., JASSI, P., S, A. N., SADAT, A., AND FEDOROVA, A. Evaluation of the Intel Core i7 Turbo Boost feature. In *Proceedings of the IEEE International Symposium on Workload Characterization* (2009).

[16] CORBET, J. Deadline scheduling for 3.14. `http://www.linuxfoundation.org/news-media/blogs/browse/2014/01/deadline-scheduling-314`, 12.04.14.

[17] DAGAND, P.-E., BAUMANN, A., AND ROSCOE, T. Filet-o-Fish: practical and dependable domain-specific languages for OS development. In *Proceedings of the 5th Workshop on Programming Languages and Operating Systems* (2009).

[18] DEPOUTOVITCH, A., AND STUMM, M. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the EuroSys Conference* (2010), pp. 181–194.

[19] DERRIN, P., ELKADUWE, D., AND ELPHINSTONE, K. *seL4 Reference Manual*. NICTA, 2006. `http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf`.

[20] ELKADUWE, D., DERRIN, P., AND ELPHINSTONE, K. Kernel design for isolation and assurance of physical memory. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems* (2008), pp. 35–40.

[21] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., AND BURGER, D. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (2011), pp. 365–376.

[22] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and automatic live update for operating systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), pp. 279–292.

[23] GLEIXNER, T., MCKENNEY, P. E., AND GUITTOT, V. Cleaning up Linux's CPU hotplug for real time and energy management. *SIGBED Rev. 9*, 4 (Nov. 2012), 49–52.

[24] HARDY, N. KeyKOS Architecture. *SIGOPS Operating Systems Review 19*, 4 (1985), 8–25.

[25] CPU hotplug. `https://wiki.linaro.org/WorkingGroups/PowerManagement/Archives/Hotplug`, 12.04.14.

[26] IPEK, E., KIRMAN, M., KIRMAN, N., AND MARTINEZ, J. F. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007), pp. 186–197.

[27] JOSHI, A. Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system. In *Proceedings of the Linux Symposium* (2010), pp. 101–108.

[28] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (2009).

[29] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: a 32-way multithreaded sparc processor. *IEEE Micro 25*, 2 (2005), 21–29.

[30] KOZUCH, M. A., KAMINSKY, M., AND RYAN, M. P. Migration without virtualization. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems* (2009), pp. 10–15.

[31] KUMAR, R., FARKAS, K. I., JOUPPI, N. P., RANGANATHAN, P., AND TULLSEN, D. M. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (2003), pp. 81–92.

[32] Real-time Linux. `https://rt.wiki.kernel.org/`, 12.04.14.

[33] LIU, R., KLUES, K., BIRD, S., HOFMEYR, S., ASANOVIĆ, K., AND KUBIATOWICZ, J. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism* (2009).

[34] MARR, D. T., DESKTOP, F. B., HILL, D. L., HINTON, G., KOUFATY, D. A., MILLER, J. A., AND UPTON, M. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal* (Feb 2002).

[35] MENZI, D. Support for heterogeneous cores for Barrelfish. Master's thesis, Department of Computer Science, ETH Zurich, July 2011.

[36] NOMURA, Y., SENZAKI, R., NAKAHARA, D., USHIO, H., KATAOKA, T., AND TANIGUCHI, H. Mint: Booting multiple Linux kernels on a multicore processor. In *Proceedings of the International Conference on Broadband and Wireless Computing, Communication and Applications* (2011), pp. 555–560.

[37] PANNEERSELVAM, S., AND SWIFT, M. M. Chameleon: Operating system support for dynamic processors. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), pp. 99–110.

[38] Popcorn Linux. `http://popcornlinux.org/`, 12.04.14.

[39] PostgreSQL. `http://www.postgresql.org/`, 12.04.14.

[40] RHODEN, B., KLUES, K., ZHU, D., AND BREWER, E. Improving per-node efficiency in the datacenter with new OS abstractions. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), pp. 25:1–25:8.

[41] SADINI, M., BARBALACE, A., RAVINDRAN, B., AND QUAGLIA, F. A Page Coherency Protocol for Popcorn Replicated-kernel Operating System. In *Proceedings of the ManyCore Architecture Research Community Symposium (MARC)* (Oct. 2013).

[42] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. EROS: A Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999), pp. 170–185.

[43] SHELDON, B. H. Popcorn Linux: enabling efficient inter-core communication in a Linux-based multikernel operating system. Master's thesis, Virginia Polytechnic Institute and State University, May 2013.

[44] SINGHANIA, A., KUZ, I., AND NEVILL, M. Capability Management in Barrelfish. Technical Note 013, Barrelfish Project, ETH Zurich, December 2013.

[45] Stress Load Generator. `http://people.seas.harvard.edu/~apw/stress/`, 12.04.14.

[46] TPC-H. `http://www.tpc.org/tpch/`, 12.04.14.

[47] VENKATESH, G., SAMPSON, J., GOULDING, N., GARCIA, S., BRYKSIN, V., LUGO-MARTINEZ, J., SWANSON, S., AND TAYLOR, M. B. Conservation Cores: Reducing the energy of mature computations. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (2010), pp. 205–218.

[48] WENTZLAFF, D., GRUENWALD III, C., BECKMANN, N., MODZELEWSKI, K., BELAY, A., YOUSEFF, L., MILLER, J., AND AGARWAL, A. An operating system for multicore and clouds: Mechanisms and implementation. In *ACM Symposium on Cloud Computing (SOCC)* (June 2010).

[49] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), pp. 79–93.

[50] ZELLWEGER, G., SCHUEPBACH, A., AND ROSCOE, T. Unifying Synchronization and Events in a Multicore OS. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems* (2012).

Jitk: A Trustworthy In-Kernel Interpreter Infrastructure

Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, Zachary Tatlock
MIT CSAIL and University of Washington

Abstract

Modern operating systems run multiple interpreters in the kernel, which enable user-space applications to add new functionality or specialize system policies. The correctness of such interpreters is critical to the overall system security: bugs in interpreters could allow adversaries to compromise user-space applications and even the kernel.

Jitk is a new infrastructure for building in-kernel interpreters that guarantee *functional correctness* as they compile user-space policies down to native instructions for execution in the kernel. To demonstrate Jitk, we implement two interpreters in the Linux kernel, BPF and INET-DIAG, which are used for network and system call filtering and socket monitoring, respectively. To help application developers write correct filters, we introduce a high-level rule language, along with a proof that Jitk correctly translates high-level rules all the way to native machine code, and demonstrate that this language can be integrated into OpenSSH with tens of lines of code. We built a prototype of Jitk on top of the CompCert verified compiler and integrated it into the Linux kernel. Experimental results show that Jitk is practical, fast, and trustworthy.

1 Introduction

Many operating systems allow user-space applications to customize and extend the kernel by downloading user-specified code into the kernel [1, 24]. One well-known example is the BSD Packet Filter (BPF) architecture [48]. With BPF, applications specify which packets they are interested in by downloading a filter program into the kernel that decides whether a packet should be dropped or forwarded to the application. For portability and safety, the kernel usually defines a simple, restricted language, and uses an *interpreter* to execute code written in that language (e.g., BPF), rather than directly downloading and executing machine code. Other notable applications of in-kernel interpreters include socket monitoring [40], dynamic tracing [7], power management [32], and system call filtering [20]. Interpreters are also used outside of kernels, such as in Bitcoin's transaction scripting [2].

As an example, consider the Seccomp subsystem [20] in the Linux kernel, which adopts the BPF language to specify what system calls a process can make. Seccomp's overall architecture is shown in Figure 1. At start-up, an application such as OpenSSH submits a BPF filter into the kernel. The kernel invokes the BPF interpreter to run the filter code against every subsequent system call. Based on the result from the interpreter, the kernel decides whether to reject or allow a system call, or kill the process altogether. Figure 2 shows the system call filter used by OpenSSH, written in the BPF language [48]. Even if an adversary later compromises the OpenSSH process, she cannot perform damaging actions, such as modifying files, as the kernel would fail the corresponding system calls, which are disallowed by the installed filter. Many other applications, such as QEMU, Chrome, Firefox, vsftpd, and Tor, secure themselves in a similar fashion.

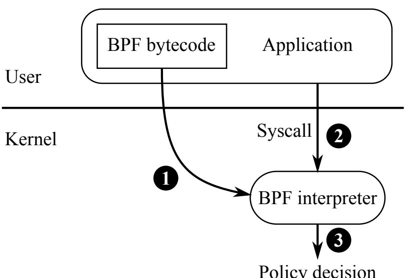


Figure 1: The architecture of the Seccomp system [20] in Linux. Application developers specify their system call policy as a BPF filter (e.g., Figure 2), in bytecode form. At start-up, the user-space application submits the filter to the kernel. The kernel invokes a BPF interpreter to evaluate the program against each subsequent system call, and decides whether to allow or reject it based on the result from the interpreter.

```
     ; load syscall number
     ld   [0]
     ; deny open() with errno = EACCES
     jeq  #SYS_open, L1, L2
L1:  ret  #RET_ERRNO|#EACCES
     ; allow getpid()
L2:  jeq  #SYS_getpid, L3, L4
L3:  ret  #RET_ALLOW
     ; allow gettimeofday()
L4:  jeq  #SYS_gettimeofday, L5, L6
L5:  ret  #RET_ALLOW
L6:  ...
     ; default: kill current process
     ret  #RET_KILL
```

Figure 2: The system call filter used in OpenSSH, in the BSD Packet Filter (BPF) language [48]. It forces the `open` system call to fail with the errno code `EACCES`, allows system calls such as `getpid` and `gettimeofday`, and kills the current process if it invokes other system calls. The `ld` instruction loads the current system call number into the accumulator register; `jeq` n, l_t, l_f is a conditional jump instruction that branches to l_t if the accumulator register is n, and otherwise branches to l_f; and `ret` terminates the filter with a return value.

```
{ default_action = Kill;
  rules = [
    { action = Errno EACCES; syscall = SYS_open };
    { action = Allow; syscall = SYS_getpid };
    { action = Allow; syscall = SYS_gettimeofday };
    ...
  ] }
```

Figure 3: OpenSSH's system call filter from Figure 2, expressed in our higher-level System Call Policy Language (SCPL).

The security of these systems critically relies on both the interpreter and user-supplied code. Since the interpreter resides in kernel-space and has full privileges, bugs in the interpreter can enable the adversary to take control of the entire system [39]. Even if a kernel compromise does not occur, bugs in the interpreter can cause it to produce the wrong result. In Seccomp, this means that the kernel may fail to stop illegal system call invocations, and thereby allow the security of the user-space application to be compromised. Finally, if user-space applications submit incorrect code to start with, such as a BPF filter that lets unintended system calls slip through, the kernel would be unable to enforce the correct policy.

Unfortunately, it is challenging to ensure that both the in-kernel interpreter and the supplied user-specified code are bug-free. First, the interpreter has a complex interface to the external world, leaving a wide range of attack vectors to adversaries who can control either user-specified code (e.g., BPF filters) or input data to the code (e.g., system call invocations), or even both. Second, the interpreter needs to handle many corner cases, such as out-of-bounds memory accesses, jumps to illegal kernel code, arithmetic errors, and infinite loops, which have historically caused problems in many systems (see §3). Third, many in-kernel interpreters employ just-in-time (JIT) compilation to convert code into native machine instructions for better performance [15, 37]; this adds another level of complexity. Fourth, there are few tools that can ensure the correctness of user-specified code.

This paper presents Jitk, a new in-kernel interpreter infrastructure that addresses these challenges through formal verification. Jitk implements a JIT that translates two languages used in the Linux kernel, BPF [48] and INET-DIAG [40], into native code, including x86, ARM, and PowerPC, and proves *functional correctness* of this translation. Jitk guarantees that the resulting native code for in-kernel execution preserves the semantics of the BPF or INET-DIAG code submitted from user space, and that the native code never performs damaging operations such as division by zero or out-of-bounds memory access.

To extend the benefits of functional correctness to user-space applications, Jitk introduces a new high-level specification language called SCPL (System Call Policy Language). Application developers can use SCPL to specify their desired system call policies using *intuitive* rules, such as "allow the `gettimeofday` system call" or "reject the `open` system call with `EACCES`." As an example, Figure 3 shows the SCPL rules that capture the BPF filter used by OpenSSH shown in Figure 2. The hope is that it is less likely for developers to make mistakes in SCPL rules than in manually written BPF filters. Jitk implements a SCPL-to-BPF compiler and a functional correctness proof from these high-level policies to native code.

The code and proof of Jitk were developed using the Coq proof assistant on top of the CompCert framework [42]. We integrated Jitk with the Linux kernel, as a drop-in replacement of its existing Seccomp subsystem. Applications like OpenSSH can run on our system without modifications, with the guarantee of the absence of interpreter bugs described in §3.

Overall, the contributions of this paper are as follows:

- The Jitk infrastructure and approach for building verified in-kernel JIT interpreters.
- A case study of real-world vulnerabilities found in BPF interpreters in several operating systems.
- A formalization of correctness and safety goals for executing user-specified policies in the kernel.
- The Jitk/BPF and Jitk/INET-DIAG verified JITs along with the formal specifications of both languages.
- The SCPL high-level language for specifying system call policy rules, along with a proof of correctness for an SCPL-to-BPF compiler.
- An evaluation of how well Jitk's formal verification prevents the vulnerabilities that have been discovered in bytecode interpreters in real-world kernels.

The rest of the paper is organized as follows. §2 discusses previous related work. §3 presents background information on the kinds of bugs that arise in in-kernel interpreters such as BPF. §4 provides an overview of Jitk's architecture and goals. §5 describes the design and proof. §6 discusses the limitations of Jitk's approach. §7 presents our prototype implementations of Jitk as applied in Seccomp and INET-DIAG in the Linux kernel. §8 evaluates Jitk's security and performance. §9 concludes.

2 Related work

Pioneering work such as seL4 [38, 54], CompCert [42], MinVisor [49], VCC [41], and Myreen's x86 JIT compiler [50] showed the promise of formal verification for building trustworthy, critical software systems, including OS kernels, compilers, and hypervisors. Jitk demonstrates how to apply formal techniques to building systems that download and execute untrusted code in a commodity kernel. Jitk leverages CompCert's compiler infrastructure and machine code semantics; alternatively, Jitk could be built on the Bedrock library [12, 13].

0	16	24	32	63
opcode	jt	jf	k	

Figure 4: A BPF instruction in bytecode format, with a fixed length of 8 bytes (64 bits). It consists of an opcode, true and false target offsets for conditional jump instructions, and a general field k.

There is a rich literature of securing and isolating faulty kernel extension through OS design, such as microkernels [18, 23, 31, 55] and exokernels [24, 34]; through the use of type-safe languages, such as SPIN (Modula-3) [1], Singularity (C#) [33], and Mirage (OCaml) [45]; through software-based fault isolation [58], such as BGI [9], LXFI [46], XFI [25], VINO [53], and SVA [17]; and through proof-carrying code [52]. These techniques focus on memory safety and kernel integrity, by isolating user-specified code from the rest of the kernel, but cannot guarantee functional correctness of the downloaded code.

Testing tools such as EXE [5], KLEE [6], and the Trinity syscall fuzzer [36] are useful for finding bugs in kernel code, and have even been applied to BPF interpreters, but they cannot guarantee bug-free code.

§3 extends Chen et al.'s earlier survey [10] with a case study of a wider range of in-kernel interpreter bugs, and the rest of the paper presents the design and implementation of Jitk that guarantees the absence of these bugs.

Our experience with Jitk suggests that it is feasible to build formally verified JITs in the kernel, on the basis of CompCert. Using a verified compiler like CompCert provides stronger assurance guarantees than some of the alternative proposals, such as integrating the LLVM infrastructure into the kernel [11].

3 Case study

Enforcing a policy in Seccomp, such as the one shown in Figure 2, involves several steps: programmers express the policy to a user-space application, which submits the policy to the kernel, which in turn relays it to an interpreter, which then either purely interprets it or compiles it to machine code for faster execution. This section summarizes representative bugs at each of these steps, using BPF as an example, and discusses the challenges of achieving correctness in this chain. Note that these bugs are general and *not* BPF-specific; they have appeared in other interpreters as well [10].

3.1 Background: the BPF virtual machine

BPF is a general-purpose virtual machine, consisting of:

- a 32-bit accumulator A,
- a 32-bit index register X,
- a scratch memory M for temporary storage (e.g., 64 bytes in the Linux kernel),
- an input packet P (the data blob for inspection), and
- an implicit program counter pc.

Opcode	Operands	Description
`ld`	$[k]$	$A \leftarrow P[k, .., k+3]$
`ja`	$\#k$	$pc \leftarrow pc + k$
`jeq`	$\#k$, `jt`, `jf`	$pc \leftarrow pc + (A = k)$? `jt` : `jf`
`div`	$\#k$	$A \leftarrow A \,/\, k$
`ret`	$\#k$	return k

Figure 5: Examples of BPF instructions. See the original BPF paper for a complete list [48].

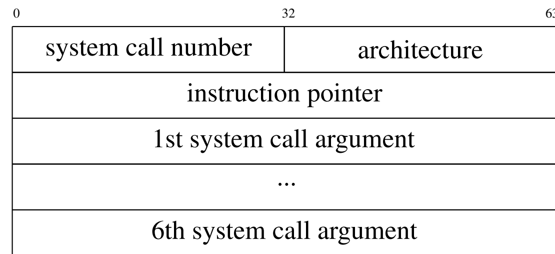


Figure 6: Input to system call filters in the Linux kernel [20], a 64-byte (512-bit) packet. It consists of the current system call number, the architecture, the instruction pointer, and up to six system call arguments.

A BPF filter is a sequence of BPF instructions, each of which has a fixed length of 8 bytes, as shown in Figure 4. It can read the input packet P, transfer data among the two registers (A and X) and the scratch memory M, perform arithmetic operations, and terminate with a 32-bit integer return value, which instructs the kernel to take further actions. Figure 5 shows examples of BPF instructions used in this paper; see the original BPF paper for a more complete list [48].

The BPF virtual machine has been successfully applied in different contexts. Its original purpose is to inspect network packets, with the return value indicating the number of bytes to accept. Its applications have gone beyond that [3, 16, 20]. For example, the Seccomp system in the Linux kernel uses BPF for system call filtering: the kernel prepares a 64-byte input packet storing the current system call arguments, as shown in Figure 6, and the return value indicates whether the kernel should fail this system call.

A *well-defined* BPF filter must end with a `ret` instruction; it can jump only forward; and it cannot perform illegal operations such as division by zero, out-of-bounds memory access, or jumping to non-existent instructions. Bugs can arise if an interpreter fails to reject illegal BPF filters, as we will show next.

3.2 Kernel-space bugs

The complex logic for executing BPF filters happens in kernel-space, where bugs can be disastrous for security. Figure 7 lists common errors that have appeared in existing BPF interpreters in Linux and BSD kernels, as detailed next.

Bug description	Examples
Kernel: control-flow errors (§3.2.1)	
jump target off by one	CVE-2014-2889 [39]
jump offset integer overflow	NetBSD PR #3366 [19], OpenBSD cvs bpf_filter.c:r1.13
Kernel: arithmetic errors (§3.2.2)	
incorrect divison-by-zero check	NetBSD PR #43185 [29], OpenBSD cvs bpf_filter.c:r1.21
incorrect reciprocal multiplication	Linux git `aee636c480` [21]
Kernel: memory errors (§3.2.3)	
buffer overflow	NetBSD PR #32198 [27], Linux git `fe15f3f1`, FreeBSD svn 182380
array index integer overflow	NetBSD PR #45751 [51], Linux git `55820ee2`, FreeBSD svn 41588
Kernel: information leak (§3.2.4)	
uninitialized read (scratch memory)	NetBSD PR #45142 [30], CVE-2010-4158 (Linux), CVE-2012-3729 (iOS)
uninitialized read (A & X registers)	Linux git `83d5b7ef99` [56]
Kernel-user interface bugs (§3.3)	
incorrect bytecode encoding/decoding	Linux git `8c482cdc` [4]
User-space bugs (§3.4)	
incorrect translation	tcpdump git `489f459b` [35], libseccomp git `cc063d8d`
incorrect optimization	tcpdump issue #38 [26], tcpdump issue #42

Figure 7: Representative bugs in BPF interpreters.

```
/* x86 code: jcc t_offset; jmp f_offset; ...
   t_offset should be increased by
   (a) 2 bytes (jmp rel8) or
   (b) 5 bytes (jmp rel32) */
jcc = /* conditional jump opcode */
if (filter[i].jf) /* BUG: should be 2 : 5 */
    t_offset += is_near(f_offset) ? 2 : 6;
EMIT_COND_JMP(jcc, t_offset);
if (filter[i].jf)
    EMIT_JMP(f_offset);
```

Figure 8: Incorrect jump target (off by one) in the BPF x86 JIT of the Linux kernel (CVE-2014-2889 [39]). The size of a `jmp` here is either 2 or 5 bytes, not 6.

```
if (BPF_OP(insn->code) == BPF_JA) {
    /* BUG: miss overflow case pc + insn->k < pc */
    if (pc + insn->k >= len)
        return 0;
}
```

Figure 9: Insufficient validation of the jump offset k [19]: given a large k, $pc + k$ will wrap around to a smaller value and bypass the check.

3.2.1 Control-flow errors

JIT interpreters need to correctly translate the control flow of a BPF filter to machine code, which is both delicate and intricate; even a tiny typo can open a new door for kernel exploits. As an example, Figure 8 shows an off-by-one bug in the x86 JIT in the Linux kernel: for a BPF conditional jump, the interpreter emits an x86 conditional jump instruction (for the true case), followed by an unconditional `jmp rel32` (for the false case), which is 5 bytes; the interpreter mistakes it as 6 bytes, and increases the offset of the conditional jump instruction by that wrong value. Consequently, the conditional jump instruction will go one byte past the target instruction, which can be an arbitrary payload controlled by an adversary [39].

```
case BPF_DIV: /* reject A / k where k = 0 */
  /* BUG: should be 0x08; 0x18 is a wrong mask */
  if ((insn->code & 0x18) == BPF_K && insn->k == 0)
      return 0;
```

Figure 10: Incorrect division-by-zero check [29]. The code uses the wrong mask `0x18`, and thus fails to reject BPF code that performs division by zero, which may lead to a kernel crash.

```
/* A / k → reciprocal_divide(A, R)
   precompute R = ((1LL << 32) + (k - 1)) / k */
u32 reciprocal_divide(u32 A, u32 R)
{
     return (u32)(((u64)A * R) >> 32);
}
```

Figure 11: Incorrect reciprocal multiplication optimization [21]. With this optimization `A/1` always produces zero, rather than `A`. The Linux kernel later disabled this optimization for BPF.

Figure 9 shows another example from BSD kernels: the interpreter needs to limit the jump offset k within the filter code, by checking if $pc + k$ exceeds the total length; otherwise an adversary can trick the kernel into executing illegal instructions. However, the interpreter misses the case where a large jump offset overflows $pc + k$ and bypasses the check.

3.2.2 Arithmetic errors

One infamous type of arithmetic errors is division by zero, which can crash the kernel if the interpreter fails to reject it. Figure 10 shows a bug where the interpreter tries to avoid that case, but performs the wrong check. Particularly, for BPF instructions A/k and A/X, one can observe their encoding difference by masking the opcode with `0x08`; the interpreter uses the wrong mask and fails to detect the case when k is zero.

```
/* BUG: k + sizeof(int32_t) can overflow */
if (k + sizeof(int32_t) > buflen)
    return 0;
A = EXTRACT_LONG(&p[k]);
```

Figure 12: Incorrect bounds check [51], which a large `k` can bypass since it overflows `k + sizeof(int32_t)` to a smaller value. A correct check is `k > buflen || sizeof(int32_t) > buflen - k`.

Optimizing arithmetic operations further complicates the situation. Figure 11 shows a bug in Linux's BPF JIT, which tries to optimize a division by a constant into a multiplication and a shift. This optimization, also used in the slab memory allocator, works well with input in that particular context (e.g., cache size as the divisor). However, this optimization is incorrect in general; for example, 65536/65537 should produce zero, but with the optimization the result becomes one. The Linux kernel has disabled this optimization for BPF [21].

3.2.3 Memory errors

An interpreter has access to two memory regions, the input packet P and the scratch memory M. It needs to correctly check the offsets of load and store instructions for both regions and reject illegal ones, which would otherwise trick the kernel into reading from or writing to memory that is out of range. The interpreter can easily miss such checks for some instructions [27], or more subtly, perform insufficient checks. Figure 12 shows an incorrect bounds check for `ld [k]`, where an adversary can use a large k to overflow and bypass the check, leading to illegal access beyond the input packet P.

3.2.4 Information leak

Since each BPF filter returns a 32-bit integer to user space, an interpreter needs to ensure that the return value is derived *only* from the input packet. In other words, it must *not* leak sensitive information from other processes nor the kernel. Several interpreters, including those in iOS (CVE-2012-3729) and Linux (CVE-2010-4158), allowed BPF filters to access uninitialized scratch memory M [30] or registers A and X [56], which could hold sensitive values from previous use. The interpreters fixed this vulnerability either by filling M, A, and X with zeros before execution, or by rejecting BPF filters that try to read these values before writing to them.

3.3 Kernel-user interface bugs

The logic at the kernel-user interface is straightforward: a user-space application encodes a BPF filter in bytecode format, as shown in Figure 4, and submits it to the kernel; the kernel decodes the bytecode and reconstructs the filter. Interestingly, there is still a possibility for programming mistakes such as copy-paste bugs [44]: for example, the Linux kernel once confused `BPF_W` with `BPF_B` for BPF bytecode encoding [4].

3.4 User-space bugs

It is tedious and error-prone to directly write BPF filters like Figure 2; for example, it requires programmers to correctly specify relative jump offsets. Many programmers instead express their policies through domain-specific tools or libraries, which provide a high-level interface for constructing filters. For example, invoking `tcpdump` with "`tcp dst port 80`" produces a 128-byte network filter for finding TCP packets sent to port 80. Applications like QEMU use the libseccomp library [22] to simplify the task of generating system call filters. These tools and libraries can submit incorrect filters to the kernel due to bugs in translating domain-specific policies into BPF filters [35], or when they try to optimize resulting filters [26].

3.5 Summary

Running user-specified code in the kernel offers flexibility and extensibility, at the price of a more vulnerable system. Achieving correctness and safety in an in-kernel interpreter is challenging: programmers can easily miss validating input for certain corner cases, or generate wrong code that is hard to notice. Many of these bugs have serious security impacts, as we have shown in Figure 7. In the next section, we will describe how to apply formal verification techniques to building Jitk, which is safe, fast, and immune to these bugs.

4 Overview

This section provides an overview of Jitk and its goals of correctly translating high-level, human-comprehensible policies to low-level native code for safe in-kernel execution. We describe Jitk in the context of the Seccomp system in Linux using the Jitk/BPF JIT, though the approach is general and not limited to BPF.

4.1 The architecture of Jitk/BPF

Figure 13 shows the architecture of Jitk/BPF. In contrast with the current Seccomp subsystem shown in Figure 1, there are three important differences.

System Call Policy Language (SCPL). Rather than manually writing BPF code, application developers can choose to specify system call policies using a high-level SCPL, which is more intuitive and helps programmers avoid mistakes in their policies. In steps 1 and 2, invoking the SCPL compiler on a SCPL program produces a corresponding BPF filter. As an example, Figure 3 shows the SCPL rules that capture the BPF filter used by OpenSSH shown in Figure 2, and our SCPL compiler will produce the latter from the former. Note that these two steps are optional; applications can still directly submit BPF bytecode to the kernel.

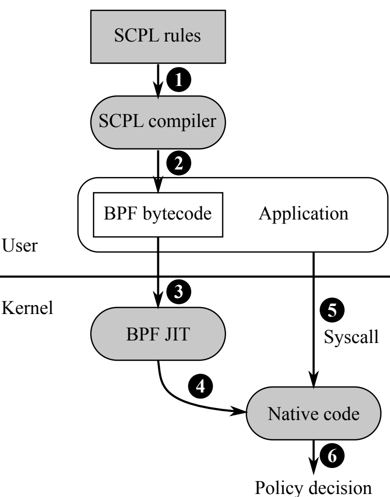


Figure 13: System overview of Jitk/BPF. Compared to the Seccomp subsystem in Linux shown in Figure 1, shaded components are introduced by Jitk. Steps are indicated with circled numbers.

JIT interpreter with a shared backend. In Jitk when the kernel accepts BPF bytecode from user space, a JIT translates the BPF filter into native code (steps 3 and 4). This native code is then executed for each system call to decide whether to allow that system call (steps 5 and 6). This JIT approach helps avoid the overhead of invoking the interpreter on each system call invocation. Jitk includes a compiler backend reused from CompCert, which is independent of BPF and can be shared among different interpreters. Our prototype implementation runs part of the JIT as a trusted user-space process (see §7).

Formal verification. The SCPL compiler and the BPF JIT are formally proven to be correct, as detailed next.

4.2 Goals

Jitk has two overall goals for enforcing user-specified policies in the kernel. First, well-behaved applications should be able to properly execute their filters in the kernel. That is, if an application developer writes down a set of SCPL rules, those rules should be correctly enforced by the kernel. We will call this the *correctness* goal. Second, it should be impossible for an adversary to misuse Jitk to "break" the kernel in any way. We will call this the *safety* goal. Jitk formalizes these goals in the form of a set of theorems and lemmas, as we will now discuss.

4.2.1 Correctness

The overall correctness goal required by an application that uses system call filtering is captured by the following end-to-end theorem:

Theorem 1 (End-to-end correctness)**.** For any system call policy p written in SCPL, if Jitk accepts it, the overall system enforces the semantics of p.

To enforce a system call policy in the kernel, Jitk needs to translate SCPL rules into BPF instructions, transmit BPF instructions from user space to the kernel, and translate BPF instructions into native code for execution in the kernel. To achieve Theorem 1, Jitk proves three lemmas that reflect this workflow.

First, the SCPL compiler must preserve the semantics of SCPL rules when generating BPF instructions:

Lemma 2 (SCPL-to-BPF semantic preservation)**.** Given a system call policy p written in SCPL, if the SCPL compiler translates it into a BPF filter f, f preserves the semantics of p:

$$\forall p : \mathit{SCPLc}(p) = \mathtt{OK}\, f \implies p \approx f.$$

Here OK means the translation is successful; $\approx$ denotes semantic preservation.

Second, a filter must be transmitted correctly from user space to the kernel. To cross the user-kernel boundary, the filter is encoded from the in-memory representation into a byte-level representation as shown in Figure 4, submitted to the kernel through a system call (e.g., `prctl` in Linux [20]), and then decoded back into the in-memory representation by the kernel's BPF JIT. The reconstructed filter in the kernel must be the same as its user-space counterpart:

Lemma 3 (User-kernel representation equivalence)**.** If a BPF filter f is encoded into bytes in user space and the bytes are decoded back to a BPF filter in kernel space, f is preserved.

$$\forall f : \mathit{encode}(f) = \mathtt{OK}\, b \implies \mathit{decode}(b) = \mathtt{OK}\, f.$$

Finally, when the JIT translates BPF instructions into native code in the kernel, the native code must preserve the semantics of the BPF instructions:

Lemma 4 (BPF-to-native semantic preservation)**.** Given a BPF filter f, if the JIT accepts it and generates native code n, n preserves the semantics of f.

$$\forall f : \mathit{jit}(f) = \mathtt{OK}\, n \implies f \approx n.$$

Jitk achieves this by first translating BPF to Cminor, an intermediate language in CompCert [42]. Jitk proves the correctness of the BPF-to-Cminor translation, and reuses Cminor-to-native from CompCert. See §5.1.2 for details.

Taken together, Lemmas 2 through 4 imply Theorem 1.

4.2.2 Safety

The safety concern is that an arbitrary user-space application should *not* be able to misuse Jitk to monopolize CPU time or to corrupt the kernel's memory. Particularly, both

native code generated by the BPF JIT and the JIT itself must be safe for in-kernel execution.

Although Theorem 1 (in particular Lemma 4) guarantees the correctness of native code with respect to given SCPL rules, it provides *no* guarantees that the generated native code will *not* cause an infinite loop or a stack overflow. The safety goal is captured by the following two theorems, which describe the temporal and spatial requirements of in-kernel execution, respectively.

Theorem 5 (Termination)**.** Given any BPF filter f, if the JIT accepts it and generates native code n, then n terminates.

$$\forall f : jit(f) = \texttt{OK}\, n \implies \texttt{terminate}(n).$$

Theorem 5 says if the JIT accepts an input BPF filter, the resulting native code must terminate, that is, the native code must come to a halt within a finite number of steps. We believe that termination is an appealing property for safety: it guarantees a bounded CPU time (i.e., no infinite loops), and *no* undefined behavior in the native code (e.g., no out-of-bounds memory access nor division by zero), with which the execution will get stuck.

Let S be a predefined parameter of the JIT, which indicates the maximum number of bytes that native code generated by the JIT can safely allocate and consume from the kernel stack.

Theorem 6 (Bounded stack usage)**.** Given any BPF filter f, if the JIT accepts it and generates native code n, n uses at most S bytes of stack.

$$\forall f : jit(f) = \texttt{OK}\, n \implies \text{any run of } n \text{ uses at most } S \text{ bytes of stack space.}$$

Theorem 6 says that if the JIT accepts an input BPF filter, the resulting native code must *never* overflow the kernel stack.

The safety of the BPF JIT itself is guaranteed by Coq. The JIT is written in Coq (see §4.3); all Coq programs are guaranteed to provide memory safety, and are guaranteed to terminate, as it is impossible to write infinite loops in Coq's Gallina language [14: §7].

4.3 Development flow

To build a trustworthy in-kernel interpreter in Jitk, developers need to prove that an implementation satisfies the correctness and safety goals as formalized in §4.2. The development workflow is shown in Figure 14.

In particular, the JIT, the encoder-decoder from user space to kernel, and the SCPL compiler are all written in Coq. For each component, Jitk's Coq source code consists of three major parts: the specification, the implementation, and the proof that the implementation matches the specification. This source code is used in two ways. First, the Coq proof checker verifies that the proof is correct. Second, Coq transforms the implementation into OCaml. The generated OCaml code is compiled, together with a small I/O stub that performs I/O and invokes the generated code, into a native executable.

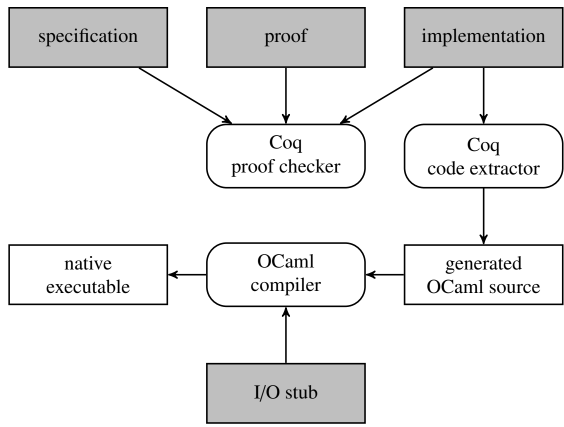


Figure 14: Development flow of Jitk using the Coq proof assistant. Shaded boxes indicate source code and proof written by developers.

5 Design

This section focuses on how the design and proofs help Jitk achieve its correctness and safety goals.

Figure 15 shows Jitk's detailed workflow and components, including the in-kernel JIT (§5.1), the high-level SCPL in user space (§5.2), the encoding-decoding across the two spaces (§5.3), and the integration of Jitk with Linux (§5.4). For each component, we will describe the specification, the implementation, and the proof.

5.1 JIT

A correct BPF JIT implementation should satisfy BPF-to-native semantic preservation (Lemma 4), termination (Theorem 5), and bounded stack usage (Theorem 6). To achieve these goals, we will start with the formal semantics of BPF (§5.1.1), and describe the design of the three key components: the translator (§5.1.2), which is responsible for transforming a BPF filter into native code, the checker (§5.1.3), which is responsible for making sure that all input filters are well-defined before being sent to the translator, and the validator (§5.1.4), which is responsible for ensuring bounded stack usage for output assembly code.

5.1.1 The specification

The specification of BPF consists of the syntax of instructions, the states, and the semantics, which is a set of state transitions among the states. The syntax mirrors the description in §3.1, which is omitted here. During execution, a BPF filter is in one of the following three states:

- initial state: a pair of current filter f and input packet P, denoted as (*Initialstate* f P);

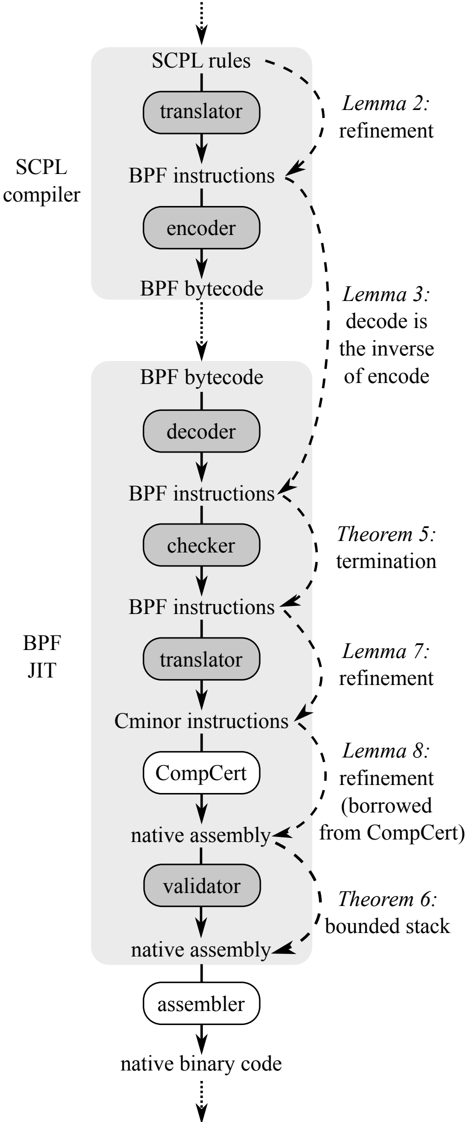


Figure 15: A detailed view of Jitk's SCPL compiler (in user space) and the BPF JIT (in kernel space).

- running state: a 6-tuple of registers A and X, scratch memory M, program counter pc, as well as f and P, denoted as ($State\ A\ X\ M\ pc\ f\ P$); and
- final state: a return value v, denoted as ($Finalstate\ v$).

A well-defined filter starts from the initial state, transitions through a set of running states, and halts in the final state. The first state transition is:

$$\begin{aligned}&(Initialstate\ f\ P)\\ \rightarrow\ &(State\ 0\ 0\ (\texttt{list_repeat}\ N\ 0)\ f\ f\ P)\end{aligned} \tag{5.1}$$

This says that at start-up, A and X are set to zero, M is initialized as N zeros, and pc is set to the start of the filter f.

The core part of the semantics is transitions between two running states after executing an instruction other than `ret`. For example, the BPF instruction "`add` k" increases the value of register A by k, and the corresponding transition is:

$$\begin{aligned}&(State\ A\ X\ M\ (\texttt{add}\ k :: pc')\ f\ P)\\ \rightarrow\ &(State\ (\texttt{Int.add}\ A\ k)\ X\ M\ pc'\ f\ P)\end{aligned}$$

Here `add` k :: pc' means that the current instruction to be executed is `add` k and the next program counter is pc'; `Int.add` is a 32-bit integer addition from CompCert. The specification says that after executing `add` k, the program transitions to a state with updated values of A and pc.

A more interesting example is a transition with a precondition. Below is the transition for BPF's unconditional jump instruction `ja` k:

$$\begin{aligned}&k < \texttt{length}\ pc' \implies\\ &(State\ A\ X\ M\ (\texttt{ja}\ k :: pc')\ f\ P)\\ \rightarrow\ &(State\ A\ X\ M\ (\texttt{skipn}\ k\ pc')\ f\ P)\end{aligned}$$

Here `length` returns the number of instructions remaining and `skipn` drops a given number of instructions. The specification says if k is less than the number of instructions remaining, then `ja` skips k instructions. Note that the specification says *nothing* about a too-large k with which `ja` could jump past the end of the filter. Therefore, such a filter is undefined, which a safe implementation like Jitk must reject.

The last state transition is to enter the final state with the return value k after executing `ret` k:

$$\begin{aligned}&(State\ A\ X\ M\ (\texttt{ret}\ k :: pc')\ f\ P)\\ \rightarrow\ &(Finalstate\ k)\end{aligned}$$

5.1.2 The translator and semantic preservation

The translator compiles a well-defined BPF filter (which passed the checker, as we will describe in §5.1.3) into native code. Our goal is to have an implementation with its correctness proof, as stated in Lemma 4. The key challenge of designing the translator is to choose an appropriate target language, which strikes a balance between the complexity of the implementation and the proof.

One approach is to directly produce low-level code such as x86 instructions from BPF, just like the existing JIT implementations in Linux and BSD kernels. The main downside of this approach, which we initially adopted, is that the big semantic gap between BPF and x86 makes it difficult to reason about the correspondence and prove properties about the target code. For example, consider translating BPF's unconditional jump `ja` k into x86's `jmp` n, where k and n are jump offsets in the corresponding languages. It is tricky to compute n (see §3.2.1); meanwhile, it is difficult to prove why n is the correct value that corresponds to k.

Jitk's translator targets Cminor, one of CompCert's intermediate languages [42]. Cminor can be considered

as an architecture-independent, simple C variant: it has only primitive types such as n-bit integers, and no pointer or struct types. The main advantage of targeting Cminor is that it retains high-level constructs, which simplifies the proof. For example, Cminor has labels and variables, with which one does not have to reason about low-level details such as jump offsets or the stack pointer.

To prove correctness of the translator, we prove that it satisfies the following property:

Lemma 7 (BPF-to-Cminor semantic preservation)**.** If the JIT translator generates Cminor code c from a BPF filter f, c preserves the semantics of f.

$$\forall f : jit_{translator}(f) = \mathtt{OK}\, c \implies f \approx c.$$

The main technique for proving Lemma 7 is by simulation [43], as used throughout CompCert. Specifically, it suffices to show that each state transition in the BPF specification shown in §5.1.1 corresponds to some state transitions in the Cminor specification during translation. We omit the details.

A second advantage of targeting Cminor is that Jitk can reuse CompCert to transform Cminor code into native assembly. CompCert currently has support for x86, ARM, and PowerPC, for which it comes with its own semantic preservation theorem:

Lemma 8 (Cminor-to-native semantic preservation)**.** If CompCert generates native code n from a Cminor program c, n preserves the semantics of c.

$$\forall c : compcert(c) = \mathtt{OK}\, n \implies c \approx n.$$

Together, Lemmas 7 and 8 imply our correctness goal, Lemma 4.

Finally, in order to transform the assembly code into native binary code, Jitk invokes a traditional assembler; CompCert does not include a provably correct assembler.

5.1.3 The checker and termination

Recall that Lemma 4 guarantees BPF-to-native semantic preservation: if an input filter terminates, the resulting native code produced by the translator also terminates. Therefore, to achieve the termination goal as stated in Theorem 5, it suffices to implement a checker that rejects all undefined input and ensures that any surviving filter terminates.

For BPF, this amounts to the following requirements: that all jump targets are forward (i.e., no loops), that all jump targets are in-bounds (i.e., not pointing past the end of the program), that all memory accesses are in-bounds (i.e., not reading or writing past the end of the input packet and the scratch memory), and that the input ends in a `ret` instruction. The combination of these rules ensures that every program that passes the checker will be well-defined, and will terminate with some return value according to BPF semantics.

We prove that the implementation of the checker satisfies the following property:

Lemma 9 (BPF termination)**.** If the JIT checker accepts a BPF filter f, then f terminates.

$$\forall f : jit_{checker}(f) = \mathtt{OK}\, f \implies \mathtt{terminate}(f).$$

Combining this with Lemma 4 gives our safety goal of termination, Theorem 5. Note that it is impossible to miss any undefined cases in the implementation of the checker, otherwise the proof of Lemma 9 would not succeed.

5.1.4 The validator and bounded stack usage

CompCert does not provide facilities for reasoning about stack bounds across transformations. In order to prove Theorem 6, one option is to extend CompCert with proposed support for tracking stack bounds [8], which would allow Jitk to prove a theorem about it.

Jitk adopts a simpler approach using the validator. As for BPF, there is only one function with a fixed number of variables and a fixed-size object (scratch memory) on the stack. The validator inspects the size in the stack frame allocation instruction at function entry in the resulting assembly code, and fails the JIT if the size is larger than a predefined S. It is then easy to prove Theorem 6, as any generated assembly code that passes the validator uses at most S bytes from the stack space.

5.2 SCPL

The design of our SCPL is inspired by the libseccomp API [22]; the key difference is that the SCPL compiler is provably correct. As shown in Figure 3, developers specify rules for matching different system calls (and optionally system call arguments), along with actions to take when those rules match (e.g., allow the system call, or reject it with a particular `errno` value). There is also a default action, if none of the rules match. The formal specification of SCPL is similar to BPF described in §5.1.1: the syntax, the states, and the state transitions. The proof of SCPL-to-BPF correctness is also similar to BPF-to-Cminor. We omit the details here.

5.3 Encoding and decoding

To ensure that BPF programs are faithfully encoded and decoded when transmitted across the user-kernel space boundary, Jitk implements an encoder and decoder, which transforms an in-memory representation of a BPF program into BPF bytecode, and back into an in-memory representation.

One option for proving the correctness of the BPF encoder and decoder would be to define semantics for BPF bytecode, as defined by byte-level sequences, and to prove equivalence between the in-memory representation (under

the semantics of in-memory BPF programs) and the byte-level representation (under the semantics of byte-level encodings). However, this approach is quite cumbersome, owing to the complexity of defining the semantics of BPF programs at the low level of byte encodings.

Instead, Jitk takes a pragmatic alternative: it proves that the decoder is the inverse of the encoder (Lemma 3). When used in combination with SCPL, this guarantees that SCPL-generated BPF bytecode will necessarily be decoded correctly by the Jitk/BPF JIT in the kernel. It does not guarantee that the encoder and decoder implemented by Jitk are compatible with any other encoding (e.g., the encoding produced by libseccomp). In practice, we believe this is not a significant problem, because the specification of the correct encoding and decoding would be of similar complexity to our current Coq encoder/decoder implementation, and our theorem (Lemma 3) provides a strong sanity check on the internal consistency of our encoder and decoder.

5.4 Linux kernel integration

The Linux kernel interacts with Jitk in two ways. The first is when an application submits a BPF filter to Seccomp through the `prctl` system call [20]: we modified the kernel to invoke Jitk's BPF JIT. The JIT translates the filter into native code; if the translation fails, indicating that the BPF filter code was not well-formed, the `prctl` system call returns an error.

The second is when an application makes a subsequent system call: we modified the kernel to check if there is already a JIT-compiled filter associated with the process; if so, the kernel treats the filter as a function pointer, invoking it with a single argument (the structure containing the system call number and arguments, shown in Figure 6).[1] The return value determines the resulting action for this system call, much as with the existing Seccomp design.

6 Discussion

There are some mistakes that Jitk's theorems cannot prevent. First, if the specifications of BPF and SCPL are buggy, then Jitk's JIT implementations can have bugs.

Another example is an overly strict checker, such as NetBSD #37663 [28], which rejected any BPF program that used the multiply instruction. Such an implementation does not violate either correctness or safety goals, since all of our theorems are conditional on our system accepting a given program. It would be possible to prove an additional theorem that required Jitk/BPF to accept certain programs; one good candidate would be a requirement that Jitk/BPF accept all BPF programs generated by the SCPL compiler.

[1] Linux kernel 3.16 or later does not require this modification anymore, as it has been changed to work in the same way.

Jitk's theorems also cannot prevent the use of Jitk's JIT for JIT spraying [47], which can make it easier to exploit memory corruption vulnerabilities in the rest of the Linux kernel. Given a formal set of requirements for a JIT to mitigate the effects of JIT spraying (e.g., ensuring that a constant in the input bytecode does not appear in the output code), it may be possible to prove that Jitk correctly implements such mitigation techniques.

Jitk's encoder/decoder can have self-consistent mistakes, in that the encoder and decoder are consistent with each other (satisfying Lemma 3), but do not match the encoding used by others for the same bytecode. We believe it is unlikely for the reasons discussed in §5.3.

Finally, Jitk assumes several additional parts are correct without a formal proof. First, the Coq proof checker is assumed to be correct. While bugs have been found in Coq, we believe Coq provides a strong degree of assurance that Jitk is trustworthy. The Coq extraction system and the OCaml compiler and runtime are also assumed to be correct. We believe this is reasonable because Coq itself is written in OCaml. That said, bugs in Coq's extraction system have been found in the past [57, 59].

Jitk's OCaml I/O stub has no proof of correctness. It is about 70 lines of code, and performs simple operations: taking input from stdin, passing it into the Coq-extracted code, and printing the results to stdout.

The rest of the kernel code, including the wrapper that invokes the Jitk JIT and that invokes the filter code produced by the JIT, is assumed to be correct. Particularly, Jitk assumes that the kernel does not trample on the JIT itself, that the kernel correctly interprets the results from the JIT and the filter, and that the kernel synthesizes an correct input packet to the filter, namely, a single pointer argument pointing to a valid memory region whose size matches the structure shown in Figure 6.

Jitk also assumes that the kernel invokes the JIT-compiled code with an appropriate calling convention. For example, on x86 the JIT-compiled code assumes that the input argument is passed on the stack, as CompCert's x86 backend uses the *cdecl* calling convention; however, the Linux kernel uses *fastcall* by default (e.g., with gcc's `-mregparm=3`), which passes the input argument in the `EAX` register. We bridged the gap using a function wrapper.

Finally, Jitk assumes that CompCert generates correct assembly code for the filter, which is a single-argument function. One technical complication is that CompCert's semantics are defined only for complete programs that take *no* arguments. This precludes even well-formed C programs with a `main` function that takes two arguments, `argc` and `argv`, for which theoretically CompCert provides no guarantees. We believe this is not a likely source of bugs in practice.

Component	Lines of code
Specifications (SCPL, BPF)	420 lines of Coq
Implementation (SCPL, BPF)	520 lines of Coq
Proof (SCPL, BPF)	2,300 lines of Coq
Extraction to OCaml	50 lines of Coq
I/O stub	70 lines of OCaml
Linux kernel changes	150 lines of C
Total	3,510 lines of code

Figure 16: Lines of code for our Jitk/BPF prototype.

7 Implementation

We have implemented a fully functional prototype of Jitk for Linux's BPF-based Seccomp system. The breakdown of our Jitk/BPF prototype (excluding the components borrowed from CompCert) in terms of lines of code is shown in Figure 16.

As mentioned in §4.3, the BPF JIT is written in Coq. We extracted the Coq implementation into OCaml code, and linked it into an executable, together with the I/O stub and the OCaml runtime. In order to run this executable for translating BPF bytecode into native binary code, we chose to put it in user space and modified the kernel to perform an upcall, using Linux's `call_usermodehelper` interface. Putting the executable with the OCaml runtime into the kernel would be doable, as demonstrated by the Mirage unikernel [45], but would unnecessarily complicate our implementation. Trusting a user-mode process running as root to produce native binary code for running in the kernel seems reasonable, given that root processes can also load arbitrary kernel modules.

Jitk supports generating x86, ARM, and PowerPC code, as CompCert provides the corresponding backends. Jitk does not support architectures that CompCert lacks, such as x86-64.

Like CompCert, Jitk relies on a conventional (not proven correct) assembler to convert textual assembly code into a native binary. Jitk uses the GNU assembler `as` for this purpose, and disables assembler-level optimizations (using the `-n` option) out of precaution.

8 Evaluation

In our evaluation, we aim to answer five questions:

- How much effort does it take to build Jitk? (§8.1)
- Does formal verification prevent the kinds of bugs that arise in practice? (§8.2)
- Does Jitk's JIT produce efficient filter code? (§8.3)
- How much effort is required to use SCPL? (§8.4)
- What is the end-to-end performance of Jitk? (§8.5)

8.1 Verification effort

The total effort to build Jitk for Seccomp was shown in Figure 16. Much of the effort went into constructing proofs. The 2,300 lines of Coq proof code consist of 650 lines of general helpers (the `crush` tactic from CPDT [14] and miscellaneous lemmas about linked lists and arithmetic), 950 lines of refinement proof (that the BPF JIT preserves semantics), 350 lines of termination proof (that programs that pass the checker are well-defined), 150 lines of encoding proof (that decoding is the inverse of encoding), and 250 lines of proof for the SCPL compiler.

To determine if this approach can be applied to another bytecode language, we implemented a JIT for the INET-DIAG interpreter from the Linux kernel. INET-DIAG has a simpler bytecode language, and the code and proof sizes were correspondingly smaller, totaling 1,590 lines of code. Overall, we believe this indicates Jitk is a practical approach for building trustworthy in-kernel interpreters.

8.2 Bug case study

To evaluate whether Jitk's formal verification does a good job of preventing bugs that arise in practice, we perform an analysis of the bugs that have been found in interpreters so far (Figure 7), and manually determine whether such a bug could have been present in an implementation that provably satisfies Jitk's theorems. Our results show that Jitk is effective at preventing these bugs.

Control-flow errors. The control flow errors described in §3.2.1, such as misaligned jump targets and overflow in computing the jump offset, cannot occur in Jitk, since Lemma 4 guarantees that all jumps in native code preserve the semantics. It is also impossible to "run off the end" of generated native code, since Theorem 5 guarantees that every BPF program that passes the checker must terminate. We found and fixed several off-by-one jump errors in our implementation while proving Lemma 7.

Arithmetic errors. If the JIT forgets to check for division by zero, the proof of Lemma 7 cannot succeed: in Cminor, division by zero is undefined, and it would be impossible to prove that the generated Cminor code refines the semantics of the BPF program. Similarly, if the JIT has an incorrect optimization, such as `reciprocal_divide`, the proof of Lemma 7 cannot proceed, either. We initially forgot to check for division by zero, and had to address it in order to complete the proof.

Memory errors. If the memory index in a BPF instruction is in-bounds, the generated Cminor instruction must access the same memory index; otherwise the proof of Lemma 7 cannot proceed. This eliminates incorrect memory indices. On the other hand, if the memory index in a BPF instruction is out-of-bounds, the BPF program is undefined, and the termination proof ensures that the checker

	BPF	x86		ARM		PowerPC	
		FreeBSD	Jitk	Linux	Jitk	Linux	Jitk
OpenSSH	312	466	274 (58.8%)	384	396 (103.1%)	452	340 (75.2%)
vsftpd	576	1,177	443 (37.6%)	616	624 (101.3%)	736	548 (74.5%)
Native Client	664	928	626 (67.5%)	828	844 (101.9%)	984	688 (69.9%)
QEMU	1,680	2,364	2,037 (86.2%)	2,048	2,156 (105.3%)	2,780	1,840 (66.2%)
Firefox	1,720	2,314	1,636 (70.7%)	2,164	2,196 (101.5%)	2,564	1,748 (68.2%)
Chrome	2,144	4,640	2,122 (45.7%)	2,380	2,348 (98.7%)	3,260	2,308 (70.8%)
Tor	3,032	6,841	2,691 (39.3%)	3,400	4,048 (119.1%)	4,308	3,304 (76.7%)

Figure 17: Comparison of sizes of native code generated by the BPF JIT of Jitk, the Linux kernel, and the FreeBSD kernel, measured in bytes. "BPF" lists the size of BPF filters, also in bytes.

must reject all such programs. Thus, out-of-bounds memory accesses are avoided.

Information leak. The initial state transition (5.1) specifies that the initial state of a BPF program contain zeroes in all registers and scratch memory locations. The proof of Lemma 4 guarantees that no generated native code can produce results inconsistent with zeroed initial memory. Note that the lemma leaves open the option of not initializing these memory locations, as long as they are not actually read by the BPF program. In fact, the CompCert compiler's optimization passes will eliminate initialization of unused memory locations in a provably correct way.

Encoding and decoding bugs. Lemma 3 guarantees that the decoder is the inverse of the encoder. This lemma's proof cannot go through if one of the encoder or decoder has a bug, as was the case in a recent Linux issue [4].

Bugs in BPF generation tools. If developers write SCPL rules and invoke the SCPL compiler to generate BPF filters, Lemma 2 guarantees the absence of incorrect filters, unlike with tcpdump or libseccomp.

8.3 Code quality

To understand the quality of native code generated by Jitk, we collect BPF filters used by popular applications shipped in Ubuntu 14.04. To extract these filters, we use `LD_PRELOAD` to intercept the `prctl` system call, used to submit system call filters to the kernel. We then compare the sizes of resulting native code produced by Jitk/BPF and by two widely used in-kernel BPF JITs, Linux and FreeBSD, as shown in Figure 17 (though Linux does not use the BPF JIT for Seccomp).

For ARM and PowerPC, we compare Jitk with Linux's BPF JIT (FreeBSD's does not support the two architectures). For x86, we compare Jitk with FreeBSD's BPF JIT (Linux's does not support x86). Also, the current Linux JIT (both ARM and PowerPC) failed on one special BPF instruction used by Seccomp; we patched it for this comparison. The results show that in addition to the correctness and safety guarantees, the quality of the native code produced by Jitk is comparable to that from existing in-kernel JITs. Particularly, Jitk generates substantially smaller code than existing in-kernel JITs on x86 and PowerPC. After inspecting the resulting assembly code, we believe the reason is that Jitk is built on top of CompCert, which performs more comprehensive and effective optimizations (e.g., common-code elimination).

8.4 SCPL

To evaluate the usability of SCPL, we translated the Seccomp policy used by OpenSSH from raw BPF operations specified by the developer into an SCPL policy. The resulting SCPL policy was 40 lines of code, parts of which were shown in Figure 3. Integrating the SCPL policy into OpenSSH required changing an additional 20 lines of OpenSSH source code, to load the BPF filter produced by the SCPL compiler, instead of using the manually written BPF filter. Overall, we believe this suggests that SCPL is easy to use in real applications.

8.5 Performance

To evaluate the performance of Seccomp with Jitk, we measured the performance of OpenSSH running on an i386 Linux system. We measured two OpenSSH configurations: one with the hand-written BPF filter, and one with the SCPL-generated filter (as described above). We also considered two kernel configurations: one using the stock Linux kernel, with its unverified BPF interpreter, and one using our modified Linux kernel that uses Jitk's BPF JIT. Since the Seccomp policy in OpenSSH applies to the process that performs user authentication, we measured the time it takes to log in via SSH and then immediately disconnect, from the same machine (i.e., measuring the time for `ssh localhost exit`), using RSA key authentication.

Figure 18 shows the results on a single-core 2.8 GHz Intel Xeon CPU with 3 GB DRAM running a 32-bit Linux 3.15-rc1 kernel. As can be seen, SCPL-generated filters perform just as well as the hand-written BPF filter in OpenSSH. Jitk's BPF JIT introduces about 20 msec of additional latency; this is due to the overhead of invoking a new process for the OCaml runtime and the assembler. We measured the time taken just to install the OpenSSH

	Stock OpenSSH BPF filter	SCPL-generated BPF filter
Stock Linux	124 msec	124 msec
Jitk BPF JIT	144 msec	144 msec

Figure 18: Time taken to login and disconnect from an OpenSSH server in different configurations; using SCPL gives the same performance as hand-written BPF filters.

BPF filter as a Seccomp policy in Linux, using the `prctl` system call; the time with the stock Linux kernel was 1 msec, and the time with Jitk's BPF JIT was 21 msec; the time to just run the BPF JIT's OCaml binary on that BPF filter is 14 msec. We believe this can be reduced further by using a persistent user-space helper process instead of spawning a new process for every BPF filter.

One benefit of Jitk's BPF JIT over a traditional interpreter is that once the BPF filter has been translated into native code, subsequent system calls can execute with lower overhead. To measure this, we used the BPF filter from OpenSSH, and measured the time to both install the BPF filter, and to perform 1,000,000 `gettimeofday` system calls (we moved `gettimeofday` to be the last system call allowed by the BPF filter). With the stock Linux BPF interpreter, this took 771 msec; with Jitk's BPF JIT, this took 691 msec; without any filter, this took 460 msec.

9 Conclusion

Jitk is a new approach for building in-kernel JIT interpreters that guarantee functional correctness using formal verification techniques. Jitk guarantees correctness through high-level policy rules in user-space applications, to lower-level BPF, across the user-kernel space boundary, and to native code in-kernel. It also guarantees termination and bounded stack usage for native code executed in-kernel. An analysis of known interpreter vulnerabilities demonstrates that Jitk prevents all classes of security vulnerabilities discovered in existing kernel interpreters. An experimental evaluation shows that Jitk's SCPL rules are easy to integrate into existing applications, and that Jitk achieves good end-to-end performance. We believe that this is a promising direction since it achieves flexibility, safety, and good performance. All of Jitk's source code is publicly available at `http://css.csail.mit.edu/jitk/`.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Gernot Heiser, for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[2] Bitcoin. Script, 2014. `https://en.bitcoin.it/wiki/Script`.

[3] D. Borkmann. net: sched: cls_bpf: add BPF-based classifier, Oct. 2013. `http://patchwork.ozlabs.org/patch/286589/`.

[4] D. Borkmann. net: filter: seccomp: fix wrong decoding of `BPF_S_ANC_SECCOMP_LD_W`, Apr. 2014. `http://patchwork.ozlabs.org/patch/339039/`.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, Alexandria, VA, Oct.–Nov. 2006.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, Dec. 2008.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 15–28, Boston, MA, June–July 2004.

[8] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–281, Edinburgh, UK, June 2014.

[9] M. Castro, M. Costa, J. P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, Big Sky, MT, Oct. 2009.

[10] H. Chen, C. Cutler, T. Kim, Y. Mao, X. Wang, N. Zeldovich, and M. F. Kaashoek. Security bugs in embedded interpreters. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, Singapore, July 2013.

[11] D. Chisnall. LLVM in the FreeBSD toolchain. In *Proceedings of AsiaBSDCon*, pages 13–20, Tokyo,

Japan, Mar. 2014.

[12] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, San Jose, CA, June 2011.

[13] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 391–402, Boston, MA, Sept. 2013.

[14] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, Nov. 2013.

[15] J. Corbet. A JIT for packet filters, Apr. 2011. https://lwn.net/Articles/437981/.

[16] J. Corbet. BPF tracing filters, Dec. 2013. https://lwn.net/Articles/575531/.

[17] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 351–366, Stevenson, WA, Oct. 2007.

[18] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 59–72, San Diego, CA, Dec. 2008.

[19] der Mouse. NetBSD PR #3366: bpf doesn't check its filter enough, Mar. 1997. http://gnats.netbsd.org/3366.

[20] W. Drewry. SECure COMPuting with filters, Jan. 2012. http://lwn.net/Articles/498231/.

[21] E. Dumazet. bpf: do not use reciprocal divide, Jan. 2014. http://patchwork.ozlabs.org/patch/311163/.

[22] J. Edge. A library for seccomp filters, Apr. 2012. http://lwn.net/Articles/494252/.

[23] K. Elphinstone and G. Heiser. From L3 to seL4: What have we learnt in 20 years of L4 microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–150, Farmington, PA, Nov. 2013.

[24] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[25] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, Seattle, WA, Nov. 2006.

[26] G. Harris. Issue #38: Another optimization bug, Dec. 2003. https://github.com/the-tcpdump-group/libpcap/issues/38.

[27] G. Harris. NetBSD PR #32198: `bpf_validate()` needs to do more checks, Nov. 2005. http://gnats.netbsd.org/32198.

[28] G. Harris. NetBSD PR #37663: `bpf_validate` rejects valid programs that use the multiply instruction, Jan. 2008. http://gnats.netbsd.org/37663.

[29] G. Harris. NetBSD PR #43185: `bpf_validate()` uses `BPF_RVAL()` when it should use `BPF_SRC()`, Apr. 2010. http://gnats.netbsd.org/43185.

[30] G. Harris. NetBSD PR #45412: `bpf_filter()` can leak kernel stack contents, July 2011. http://gnats.netbsd.org/45412.

[31] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the 2009 IEEE Dependable Systems and Networks Conference*, pages 33–42, Lisbon, Portugal, June–July 2009.

[32] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification, Dec. 2011. http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf.

[33] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, Oct. 2005.

[34] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 52–65, Saint-Malo, France, Oct. 1997.

[35] B. Keats. TCPDUMP 3.9.4 under Fedora Core 5 seems to generate the wrong BPF for `DLT_PRISM_HEADER`, Aug. 2006. http://seclists.org/tcpdump/2006/q3/37.

[36] M. Kerrisk. LCA: The Trinity fuzz tester, Feb. 2013. https://lwn.net/Articles/536173/.

[37] J.-u. Kim. Add experimental BPF Just-In-Time compiler for amd64 and i386, Dec. 2005. http://docs.freebsd.org/cgi/mid.cgi?200512060258.jB62wCnk084452.

[38] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, Oct. 2009.

[39] M. Koetter. Linux 3.0 bpf jit x86_64 exploit, Dec. 2011. http://carnivore.it/2011/12/27/linux_3.0_bpf_jit_x86_64_exploit.

[40] A. Kuznetosv. SS utility: Quick intro, Sept. 2001. http://www.cyberciti.biz/files/ss.html.

[41] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of 16th International Symposium on Formal Methods*, pages 806–809, Eindhoven, the Netherlands, Nov. 2009.

[42] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[43] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, Dec. 2009.

[44] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 289–302, San Francisco, CA, Dec. 2004.

[45] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–472, Houston, TX, Mar. 2013.

[46] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 115–128, Cascais, Portugal, Oct. 2011.

[47] K. McAllister. Attacking hardened Linux systems with kernel JIT spraying, Nov. 2012. http://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html.

[48] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 259–270, San Diego, CA, Jan. 1993.

[49] M. McCoyd, R. Krug, D. Goel, M. Dahlin, and W. Young. Building a hypervisor on a formally verified protection layer. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 5069–5078, Maui, HI, Jan. 2013.

[50] M. O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, pages 107–118, Madrid, Spain, Jan. 2011.

[51] A. Nasonov. NetBSD PR #45751: No overflow check in BPF_LD|BPF_ABS, Dec. 2011. http://gnats.netbsd.org/45751.

[52] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, WA, Oct. 1996.

[53] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–227, Seattle, WA, Oct. 1996.

[54] T. Sewell, M. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 471–482, Seattle, WA, June 2013.

[55] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185, Kiawah Island, SC, Dec. 1999.

[56] A. Starovoitov. net: filter: initialize A and X registers, Apr. 2014. http://patchwork.ozlabs.org/patch/341693/.

[57] F. Tuong. Bug 2570 - in extraction optimization, a eta-reduction leads to a not generalizable '_a, July 2011. https://coq.inria.fr/bugs/show_bug.cgi?id=2570.

[58] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, Dec. 1993.

[59] R. Zumkeller. Bug 843 - extraction breaks module typing, Aug. 2004. https://coq.inria.fr/bugs/show_bug.cgi?id=843.

IX: A Protected Dataplane Operating System for High Throughput and Low Latency

Adam Belay[1] George Prekas[2] Ana Klimovic[1] Samuel Grossman[1]
Christos Kozyrakis[1] Edouard Bugnion[2]
[1]Stanford University [2]EPFL

Abstract

The conventional wisdom is that aggressive networking requirements, such as high packet rates for small messages and microsecond-scale tail latency, are best addressed outside the kernel, in a user-level networking stack. We present IX, a *dataplane operating system* that provides high I/O performance, while maintaining the key advantage of strong protection offered by existing kernels. IX uses hardware virtualization to separate management and scheduling functions of the kernel (control plane) from network processing (dataplane). The dataplane architecture builds upon a native, zero-copy API and optimizes for both bandwidth and latency by dedicating hardware threads and networking queues to dataplane instances, processing bounded batches of packets to completion, and by eliminating coherence traffic and multi-core synchronization. We demonstrate that IX outperforms Linux and state-of-the-art, user-space network stacks significantly in both throughput and end-to-end latency. Moreover, IX improves the throughput of a widely deployed, key-value store by up to 3.6× and reduces tail latency by more than 2×.

1 Introduction

Datacenter applications such as search, social networking, and e-commerce platforms are redefining the requirements for systems software. A single application can consist of hundreds of software services, deployed on thousands of servers, creating a need for networking stacks that provide more than high streaming performance. The new requirements include high packet rates for short messages, microsecond-level responses to remote requests with tight tail latency guarantees, and support for high connection counts and churn [2, 14, 46]. It is also important to have a strong protection model and be elastic in resource usage, allowing other applications to use any idling resources in a shared cluster.

The conventional wisdom is that there is a basic mismatch between these requirements and existing networking stacks in commodity operating systems. Consequently, some systems bypass the kernel and implement the networking stack in user-space [29, 32, 40, 59, 61]. While kernel bypass eliminates context switch overheads, on its own it does not eliminate the difficult tradeoffs between high packet rates and low latency (see §5.2). Moreover, user-level networking suffers from lack of protection. Application bugs and crashes can corrupt the networking stack and impact other workloads. Other systems go a step further by also replacing TCP/IP with RDMA in order to offload network processing to specialized adapters [17, 31, 44, 47]. However, such adapters must be present at both ends of the connection and can only be used within the datacenter.

We propose IX, an operating system designed to break the 4-way tradeoff between high throughput, low latency, strong protection, and resource efficiency. Its architecture builds upon the lessons from high performance middleboxes, such as firewalls, load-balancers, and software routers [16, 34]. IX separates the control plane, which is responsible for system configuration and coarse-grain resource provisioning between applications, from the dataplanes, which run the networking stack and application logic. IX leverages Dune and virtualization hardware to run the dataplane kernel and the application at distinct protection levels and to isolate the control plane from the dataplane [7]. In our implementation, the control plane is the full Linux kernel and the dataplanes run as protected, library-based operating systems on dedicated hardware threads.

The IX dataplane allows for networking stacks that optimize for both bandwidth and latency. It is designed around a native, zero-copy API that supports processing of bounded batches of packets to completion. Each dataplane executes all network processing stages for a batch of packets in the dataplane kernel, followed by the associ-

ated application processing in user mode. This approach amortizes API overheads and improves both instruction and data locality. We set the batch size adaptively based on load. The IX dataplane also optimizes for multi-core scalability. The network adapters (NICs) perform flow-consistent hashing of incoming traffic to distinct queues. Each dataplane instance exclusively controls a set of these queues and runs the networking stack and a single application without the need for synchronization or coherence traffic during common case operation. The IX API departs from the POSIX API, and its design is guided by the commutativity rule [12]. However, the `libix` user-level library includes an event-based API similar to the popular `libevent` library [51], providing compatibility with a wide range of existing applications.

We compare IX with a TCP/IP dataplane against Linux 3.16.1 and mTCP, a state-of-the-art user-level TCP/IP stack [29]. On a 10GbE experiment using short messages, IX outperforms Linux and mTCP by up to 10× and 1.9× respectively for throughput. IX further scales to a 4x10GbE configuration using a single multi-core socket. The unloaded uni-directional latency for two IX servers is 5.7µs, which is 4× better than between standard Linux kernels and an order of magnitude better than mTCP, as both trade-off latency for throughput. Our evaluation with memcached, a widely deployed key-value store, shows that IX improves upon Linux by up to 3.6× in terms of throughput at a given 99th percentile latency bound, as it can reduce kernel time, due essentially to network processing, from $\sim$ 75% with Linux to $< 10\%$ with IX.

IX demonstrates that, by revisiting networking APIs and taking advantage of modern NICs and multi-core chips, we can design systems that achieve high throughput and low latency and robust protection and resource efficiency. It also shows that, by separating the small subset of performance-critical I/O functions from the rest of the kernel, we can architect radically different I/O systems and achieve large performance gains, while retaining compatibility with the huge set of APIs and services provided by a modern OS like Linux.

The rest of the paper is organized as follows. §2 motivates the need for a new OS architecture. §3 and §4 present the design principles and implementation of IX. §5 presents the quantitative evaluation. §6 and §7 discuss open issues and related work.

2 Background and Motivation

Our work focuses on improving operating systems for applications with aggressive networking requirements running on multi-core servers.

2.1 Challenges for Datacenter Applications

Large-scale, datacenter applications pose unique challenges to system software and their networking stacks:

Microsecond tail latency: To enable rich interactions between a large number of services without impacting the overall latency experienced by the user, it is essential to reduce the latency for some service requests to a few tens of µs [3, 54]. Because each user request often involves hundreds of servers, we must also consider the long tail of the latency distributions of RPC requests across the datacenter [14]. Although tail-tolerance is actually an end-to-end challenge, the system software stack plays a significant role in exacerbating the problem [36]. Overall, each service node must ideally provide tight bounds on the 99th percentile request latency.

High packet rates: The requests and, often times, the replies between the various services that comprise a datacenter application are quite small. In Facebook's `memcached` service, for example, the vast majority of requests uses keys shorter than 50 bytes and involves values shorter than 500 bytes [2], and each node can scale to serve millions of requests per second [46].

The high packet rate must also be sustainable under a large number of concurrent connections and high connection churn [23]. If the system software cannot handle large connection counts, there can be significant implications for applications. The large connection count between application and `memcached` servers at Facebook made it impractical to use TCP sockets between these two tiers, resulting in deployments that use UDP datagrams for `get` operations and an aggregation proxy for `put` operations [46].

Protection: Since multiple services commonly share servers in both public and private datacenters [14, 25, 56], there is need for isolation between applications. The use of kernel-based or hypervisor-based networking stacks largely addresses the problem. A trusted network stack can firewall applications, enforce access control lists (ACLs), and implement limiters and other policies based on bandwidth metering.

Resource efficiency: The load of datacenter applications varies significantly due to diurnal patterns and spikes in user traffic. Ideally, each service node will use the fewest resources (cores, memory, or IOPS) needed to satisfy packet rate and tail latency requirements at any point. The remaining server resources can be allocated to other applications [15, 25] or placed into low power mode for energy efficiency [4]. Existing operating systems can support such resource usage policies [36, 38].

2.2 The Hardware – OS Mismatch

The wealth of hardware resources in modern servers should allow for low latency and high packet rates for datacenter applications. A typical server includes one or two processor sockets, each with eight or more multithreaded cores and multiple, high-speed channels to DRAM and PCIe devices. Solid-state drives and PCIe-based Flash storage are also increasingly popular. For networking, 10 GbE NICs and switches are widely deployed in datacenters, with 40 GbE and 100 GbE technologies right around the corner. The combination of tens of hardware threads and 10 GbE NICs should allow for rates of 15M packets/sec with minimum sized packets. We should also achieve 10–20µs round-trip latencies given 3µs latency across a pair of 10 GbE NICs, one to five switch crossings with cut-through latencies of a few hundred ns each, and propagation delays of 500ns for 100 meters of distance within a datacenter.

Unfortunately, commodity operating systems have been designed under very different hardware assumptions. Kernel schedulers, networking APIs, and network stacks have been designed under the assumptions of multiple applications sharing a single processing core and packet inter-arrival times being many times higher than the latency of interrupts and system calls. As a result, such operating systems trade off both latency and throughput in favor of fine-grain resource scheduling. Interrupt coalescing (used to reduce processing overheads), queuing latency due to device driver processing intervals, the use of intermediate buffering, and CPU scheduling delays frequently add up to several hundred µs of latency to remote requests. The overheads of buffering and synchronization needed to support flexible, fine-grain scheduling of applications to cores increases CPU and memory system overheads, which limits throughput. As requests between service tiers of datacenter applications often consist of small packets, common NIC hardware optimizations, such as TCP segmentation and receive side coalescing, have a marginal impact on packet rate.

2.3 Alternative Approaches

Since the network stacks within commodity kernels cannot take advantage of the abundance of hardware resources, a number of alternative approaches have been suggested. Each alternative addresses a subset, but not all, of the requirements for datacenter applications.

User-space networking stacks: Systems such as OpenOnload [59], mTCP [29], and Sandstorm [40] run the entire networking stack in user-space in order to eliminate kernel crossing overheads and optimize packet processing without incurring the complexity of kernel modifications. However, there are still tradeoffs between packet rate and latency. For instance, mTCP uses dedicated threads for the TCP stack, which communicate at relatively coarse granularity with application threads. This aggressive batching amortizes switching overheads at the expense of higher latency (see §5). It also complicates resource sharing as the network stack must use a large number of hardware threads regardless of the actual load. More importantly, security tradeoffs emerge when networking is lifted into the user-space and application bugs can corrupt the networking stack. For example, an attacker may be able to transmit raw packets (a capability that normally requires root privileges) to exploit weaknesses in network protocols and impact other services [8]. It is difficult to enforce any security or metering policies beyond what is directly supported by the NIC hardware.

Alternatives to TCP: In addition to kernel bypass, some low-latency object stores rely on RDMA to offload protocol processing on dedicated Infiniband host channel adapters [17, 31, 44, 47]. RDMA can reduce latency, but requires that specialized adapters be present at both ends of the connection. Using commodity Ethernet networking, Facebook's memcached deployment uses UDP to avoid connection scalability limitations [46]. Even though UDP is running in the kernel, reliable communication and congestion management are entrusted to applications.

Alternatives to POSIX API: MegaPipe replaces the POSIX API with lightweight sockets implemented with in-memory command rings [24]. This reduces some software overheads and increases packet rates, but retains all other challenges of using an existing, kernel-based networking stack.

OS enhancements: Tuning kernel-based stacks provides incremental benefits with superior ease of deployment. Linux `SO_REUSEPORT` allows multi-threaded applications to accept incoming connections in parallel. Affinity-accept reduces overheads by ensuring all processing for a network flow is affinitized to the same core [49]. Recent Linux Kernels support a busy polling driver mode that trades increased CPU utilization for reduced latency [27], but it is not yet compatible with `epoll`. When microsecond latencies are irrelevant, properly tuned stacks can maintain millions of open connections [66].

3 IX Design Approach

The first two requirements in §2.1 — microsecond latency and high packet rates — are not unique to datacenter applications. These requirements have been addressed in the design of middleboxes such as firewalls, load-balancers, and software routers [16, 34] by integrating the network-

ing stack and the application into a single *dataplane*. The two remaining requirements — protection and resource efficiency — are not addressed in middleboxes because they are single-purpose systems, not exposed directly to users.

Many middlebox dataplanes adopt design principles that differ from traditional OSes. First, they *run each packet to completion*. All network protocol and application processing for a packet is done before moving on to the next packet, and application logic is typically intermingled with the networking stack without any isolation. By contrast, a commodity OS decouples protocol processing from the application itself in order to provide scheduling and flow control flexibility. For example, the kernel relies on device and soft interrupts to context switch from applications to protocol processing. Similarly, the kernel's network stack will generate TCP `ACKs` and slide its receive window even when the application is not consuming data, up to an extent. Second, middlebox dataplanes optimize for *synchronization-free operation* in order to scale well on many cores. Network flows are distributed into distinct queues via flow-consistent hashing and common case packet processing requires no synchronization or coherence traffic between cores. By contrast, commodity OSes tend to rely heavily on coherence traffic and are structured to make frequent use of locks and other forms of synchronization.

IX extends the dataplane architecture to support untrusted, general-purpose applications and satisfy all requirements in §2.1. Its design is based on the following key principles:

Separation and protection of control and data plane: IX separates the control function of the kernel, responsible for resource configuration, provisioning, scheduling, and monitoring, from the dataplane, which runs the networking stack and application logic. Like a conventional OS, the control plane multiplexes and schedules resources among dataplanes, but in a coarse-grained manner in space and time. Entire cores are dedicated to dataplanes, memory is allocated at large page granularity, and NIC queues are assigned to dataplane cores. The control plane is also responsible for elastically adjusting the allocation of resources between dataplanes.

The separation of control and data plane also allows us to consider radically different I/O APIs, while permitting other OS functionality, such as file system support, to be passed through to the control plane for compatibility. Similar to the Exokernel [19], each dataplane runs a single application in a single address space. However, we use modern virtualization hardware to provide three-way isolation between the control plane, the dataplane, and untrusted user code [7]. Dataplanes have capabilities similar to guest OSes in virtualized systems. They manage their own address translations, on top of the address space provided by the control plane, and can protect the networking stack from untrusted application logic through the use of privilege rings. Moreover, dataplanes are given direct pass-through access to NIC queues through memory mapped I/O.

Run to completion with adaptive batching: IX dataplanes run to completion all stages needed to receive and transmit a packet, interleaving protocol processing (kernel mode) and application logic (user mode) at well-defined transition points. Hence, there is no need for intermediate buffering between protocol stages or between application logic and the networking stack. Unlike previous work that applied a similar approach to eliminate receive livelocks during congestion periods [45], IX uses run to completion during all load conditions. Thus, we are able to use polling and avoid interrupt overhead in the common case by dedicating cores to the dataplane. We still rely on interrupts as a mechanism to regain control, for example, if application logic is slow to respond. Run to completion improves both message throughput and latency because successive stages tend to access many of the same data, leading to better data cache locality.

The IX dataplane also makes extensive use of batching. Previous systems applied batching at the system call boundary [24, 58] and at the network API and hardware queue level [29]. We apply batching in every stage of the network stack, including but not limited to system calls and queues. Moreover, we use batching *adaptively* as follows: (i) we never wait to batch requests and batching only occurs in the presence of congestion; (ii) we set an upper bound on the number of batched packets. Using batching only on congestion allows us to minimize the impact on latency, while bounding the batch size prevents the live set from exceeding cache capacities and avoids transmit queue starvation. Batching improves packet rate because it amortizes system call transition overheads and improves instruction cache locality, prefetching effectiveness, and branch prediction accuracy. When applied adaptively, batching also decreases latency because these same efficiencies reduce head-of-line blocking.

The combination of bounded, adaptive batching and run to completion means that queues for incoming packets can build up only at the NIC edge, before packet processing starts in the dataplane. The networking stack sends acknowledgments to peers only as fast as the application can process them. Any slowdown in the application-processing rate quickly leads to shrinking windows in peers. The dataplane can also monitor queue depths at

the NIC edge and signal the control plane to allocate additional resources for the dataplane (more hardware threads, increased clock frequency), notify peers explicitly about congestion (e.g., via ECN [52]), and make policy decisions for congestion management (e.g., via RED [22]).

Native, zero-copy API with explicit flow control: We do not expose or emulate the POSIX API for networking. Instead, the dataplane kernel and the application communicate at coordinated transition points via messages stored in memory. Our API is designed for true zero-copy operation in both directions, improving both latency and packet rate. The dataplane and application cooperatively manage the message buffer pool. Incoming packets are mapped read-only into the application, which may hold onto message buffers and return them to the dataplane at a later point. The application sends to the dataplane scatter/gather lists of memory locations for transmission but, since contents are not copied, the application must keep the content immutable until the peer acknowledges reception. The dataplane enforces flow control correctness and may trim transmission requests that exceed the available size of the sliding window, but the application controls transmit buffering.

Flow consistent, synchronization-free processing: We use multi-queue NICs with receive-side scaling (RSS [43]) to provide flow-consistent hashing of incoming traffic to distinct hardware queues. Each hardware thread (hyperthread) serves a single receive and transmit queue per NIC, eliminating the need for synchronization and coherence traffic between cores in the networking stack. The control plane establishes the mapping of RSS flow groups to queues to balance the traffic among the hardware threads. Similarly, memory management is organized in distinct pools for each hardware thread. The absence of a POSIX socket API eliminates the issue of the shared file descriptor namespace in multithreaded applications [12]. Overall, the IX dataplane design scales well with the increasing number of cores in modern servers, which improves both packet rate and latency. This approach does not restrict the memory model for applications, which can take advantage of coherent, shared memory to exchange information and synchronize between cores.

4 IX Implementation

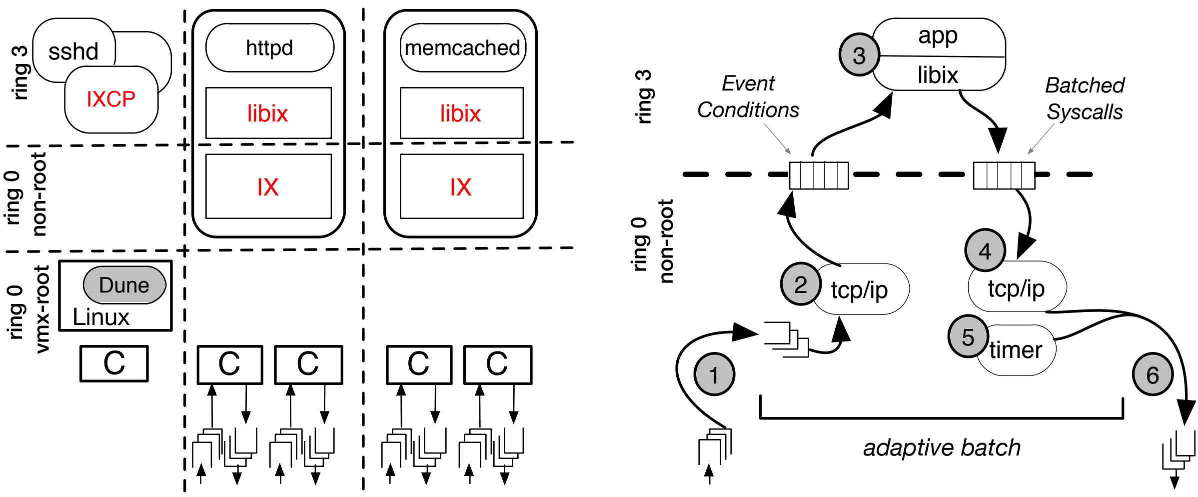
4.1 Overview

Fig. 1a presents the IX architecture, focusing on the separation between the control plane and the multiple dataplanes. The hardware environment is a multi-core server with one or more multi-queue NICs with RSS support. The IX control plane consists of the full Linux kernel and `IXCP`, a user-level program. The Linux kernel initializes PCIe devices, such as the NICs, and provides the basic mechanisms for resource allocation to the dataplanes, including cores, memory, and network queues. Equally important, Linux provides system calls and services that are necessary for compatibility with a wide range of applications, such as file system and signal support. `IXCP` monitors resource usage and dataplane performance and implements resource allocation policies. The development of efficient allocation policies involves understanding difficult tradeoffs between dataplane performance, energy proportionality, and resource sharing between co-located applications as their load varies over time. We leave the design of such policies to future work and focus primarily on the IX dataplane architecture.

We run the Linux kernel in VMX root ring 0, the mode typically used to run hypervisors in virtualized systems [62]. We use the Dune module within Linux to enable dataplanes to run as application-specific OSes in VMX non-root ring 0, the mode typically used to run guest kernels in virtualized systems [7]. Applications run in VMX non-root ring 3, as usual. This approach provides dataplanes with direct access to hardware features, such as page tables and exceptions, and pass-through access to NICs. Moreover, it provides full, three-way protection between the control plane, dataplanes, and untrusted application code.

Each IX dataplane supports a single, multithreaded application. For instance, Fig. 1a shows one dataplane for a multi-threaded `memcached` server and another dataplane for a multi-threaded `httpd` server. The control plane allocates resources to each dataplane in a coarse-grained manner. Core allocation is controlled through real-time priorities and `cpusets`; memory is allocated in large pages; each NIC hardware queue is assigned to a single dataplane. This approach avoids the overheads and unpredictability of fine-grained time multiplexing of resources between demanding applications [36].

Each IX dataplane operates as a single address-space OS and supports two thread types within a shared, user-level address space: (i) *elastic threads* which interact with the IX dataplane to initiate and consume network I/O and (ii) *background threads*. Both elastic and background threads can issue arbitrary POSIX system calls that are intermediated and validated for security by the dataplane before being forwarded to the Linux kernel. Elastic threads are expected to *not* issue blocking calls because of the adverse impact on network behavior resulting from delayed packet processing. Each elastic thread makes exclusive use of a core or hardware thread allocated



(a) Protection and separation of control and data plane.

(b) Interleaving of protocol processing and application execution.

Figure 1: The IX dataplane operating system.

to the dataplane in order to achieve high performance with predictable latency. In contrast, multiple background threads may timeshare an allocated hardware thread. For example, if an application were allocated four hardware threads, it could use all of them as elastic threads to serve external requests or it could temporarily transition to three elastic threads and use one background thread to execute tasks such as garbage collection. When the control plane revokes or allocates an additional hardware thread using a protocol similar to the one in Exokernel [19], the dataplane adjusts its number of elastic threads.

4.2 The IX Dataplane

We now discuss the IX dataplane in more detail. It differs from a typical kernel in that it is specialized for high performance network I/O and runs only a single application, similar to a library OS but with memory isolation. However, our dataplane still provides many familiar kernel-level services.

For memory management, we accept some internal memory fragmentation in order to reduce complexity and improve efficiency. All hot-path data objects are allocated from per hardware thread memory pools. Each memory pool is structured as arrays of identically sized objects, provisioned in page-sized blocks. Free objects are tracked with a simple free list, and allocation routines are inlined directly into calling functions. *Mbufs*, the storage object for network packets, are stored as contiguous chunks of bookkeeping data and MTU-sized buffers, and are used for both receiving and transmitting packets.

The dataplane also manages its own virtual address translations, supported through nested paging. In contrast to contemporary OSes, it uses exclusively large pages (2MB). We favor large pages due to their reduced address translation overhead [5, 7] and the relative abundance of physical memory resources in modern servers. The dataplane maintains only a single address space; kernel pages are protected with supervisor bits. We deliberately chose not to support swappable memory in order to avoid adding performance variability.

We provide a hierarchical timing wheel implementation for managing network timeouts, such as TCP retransmissions [63]. It is optimized for the common case where most timers are canceled before they expire. We support extremely high-resolution timeouts, as low as 16 µs, which has been shown to improve performance during TCP incast congestion [64].

Our current IX dataplane implementation is based on Dune and requires the VT-x virtualization features available on Intel x86-64 systems [62]. However, it could be ported to any architecture with virtualization support, such as ARM, SPARC, and Power. It also requires one or more Intel 82599 chipset NICs, but it is designed to easily support additional drivers. The IX dataplane currently consists of 39K SLOC [67] and leverages some existing codebases: 41% is derived from the DPDK variant of the Intel NIC device driver [28], 26% from the lwIP TCP/IP stack [18], and 15% from the Dune library. We did not use the remainder of the DPDK framework, and all three code bases are highly modified for IX. The rest is approximately 7K SLOC of new code. We chose lwIP as a starting point for TCP/IP processing because of its modularity and its maturity as a RFC-compliant, feature-rich networking stack. We implemented our own RFC-

System Calls (batched)		
Type	**Parameters**	**Description**
connect	cookie, dst_IP, dst_port	Opens a connection
accept	handle, cookie	Accepts a connection
sendv	handle, scatter_gather_array	Transmits a scatter-gather array of data
recv_done	handle, bytes_acked	Advances the receive window and frees memory buffers
close	handle	Closes or rejects a connection

Event Conditions		
Type	**Parameters**	**Description**
knock	handle, src_IP, src_port	A remotely initiated connection was opened
connected	cookie, outcome	A locally initiated connection finished opening
recv	cookie, mbuf_ptr, mbuf_len	A message buffer was received
sent	cookie, bytes_sent, window_size	A send completed and/or the window size changed
dead	cookie, reason	A connection was terminated

Table 1: The IX dataplane system call and event condition API.

compliant support for UDP, ARP, and ICMP. Since lwIP was optimized for memory efficiency in embedded environments, we had to radically change its internal data structures for multi-core scalability and fine-grained timer management. However, we did not yet optimize the lwIP code for performance. Hence, the results of §5 have room for improvement.

4.3 Dataplane API and Operation

The elastic threads of an application interact with the IX dataplane through three asynchronous, non-blocking mechanisms summarized in Table 1: they issue *batched systems calls* to the dataplane; they consume *event conditions* generated by the dataplane; and they have direct, but safe, access to (*mbuf*s) containing incoming payloads. The latter allows for zero-copy access to incoming network traffic. The application can hold on to mbufs until it asks the dataplane to release them via the `recv_done` batched system call.

Both batched system calls and event conditions are passed through arrays of shared memory, managed by the user and the kernel respectively. IX provides an unbatched system call (`run_io`) that yields control to the kernel and initiates a new run to completion cycle. As part of the cycle, the kernel overwrites the array of batched system call requests with corresponding return codes and populates the array of event conditions. The handles defined in Table 1 are kernel-level flow identifiers. Each handle is associated with a cookie, an opaque value provided by the user at connection establishment to enable efficient user-level state lookup [24].

IX differs from POSIX sockets in that it directly exposes flow control conditions to the application. The `sendv` system call does not return the number of bytes buffered. Instead, it returns the number of bytes that were accepted and sent by the TCP stack, as constrained by correct TCP sliding window operation. When the receiver acknowledges the bytes, a `sent` event condition informs the application that it is possible to send more data. Thus, send window-sizing policy is determined entirely by the application. By contrast, conventional OSes buffer send data beyond raw TCP constraints and apply flow control policy inside the kernel.

We built a user-level library, called `libix`, which abstracts away the complexity of our low-level API. It provides a compatible programming model for legacy applications and significantly simplifies the development of new applications. `libix` currently includes a very similar interface to `libevent` and non-blocking POSIX socket operations. It also includes new interfaces for zero-copy read and write operations that are more efficient, at the expense of requiring changes to existing applications.

`libix` automatically coalesces multiple write requests into single `sendv` system calls during each batching round. This improves locality, simplifies error handling, and ensures correct behavior, as it preserves the data stream order even if a transmit fails. Coalescing also facilitates transmit flow control because we can use the transmit vector (the argument to `sendv`) to keep track of outgoing data buffers and, if necessary, reissue writes when the transmit window has more available space, as notified by the `sent` event condition. Our buffer sizing policy is currently very basic; we enforce a maximum pending send byte limit, but we plan to make this more dynamic in the future [21].

Fig. 1b illustrates the run-to-completion operation for an elastic thread in the IX dataplane. NIC receive buffers are mapped in the server's main memory and the NIC's receive descriptor ring is filled with a set of buffer descriptors that allow it to transfer incoming packets using DMA. The elastic thread (1) polls the receive descriptor ring and

potentially posts fresh buffer descriptors to the NIC for use with future incoming packets. The elastic thread then (2) processes a bounded number of packets through the TCP/IP networking stack, thereby generating event conditions. Next, the thread (3) switches to the user-space application, which consumes all event conditions. Assuming that the incoming packets include remote requests, the application processes these requests and responds with a batch of system calls. Upon return of control from user-space, the thread (4) processes all batched system calls, and in particular the ones that direct outgoing TCP/IP traffic. The thread also (5) runs all kernel timers in order to ensure compliant TCP behavior. Finally (6), it places outgoing Ethernet frames in the NIC's transmit descriptor ring for transmission, and it notifies the NIC to initiate a DMA transfer for these frames by updating the transmit ring's tail register. In a separate pass, it also frees any buffers that have finished transmitting, based on the transmit ring's head position, potentially generating `sent` event conditions. The process repeats in a loop until there is no network activity. In this case, the thread enters a quiescent state which involves either hyperthread-friendly polling or optionally entering a power efficient C-state, at the cost of some additional latency.

4.4 Multi-core Scalability

The IX dataplane is optimized for multi-core scalability, as elastic threads operate in a synchronization and coherence free manner in the common case. This is a stronger requirement than lock-free synchronization, which requires expensive atomic instructions even when a single thread is the primary consumer of a particular data structure [13]. This is made possible through a set of conscious design and implementation tradeoffs.

First, system call implementations can only be synchronization-free if the API itself is commutative [12]. The IX API is commutative between elastic threads. Each elastic thread has its own flow identifier namespace, and an elastic thread cannot directly perform operations on flows that it does not own.

Second, the API implementation is carefully optimized. Each elastic thread manages its own memory pools, hardware queues, event condition array, and batched system call array. The implementation of event conditions and batched system calls benefits directly from the explicit, cooperative control transfers between IX and the application. Since there is no concurrent execution by producer and consumer, event conditions and batched system calls are implemented without synchronization primitives based on atomics.

Third, the use of flow-consistent hashing at the NICs ensures that each elastic thread operates on a disjoint subset of TCP flows. Hence, no synchronization or coherence occurs during the processing of incoming requests for a server application. For client applications with outbound connections, we need to ensure that the reply is assigned to the same elastic thread that made the request. Since we cannot reverse the Toeplitz hash used by RSS [43], we simply probe the ephemeral port range to find a port number that would lead to the desired behavior. Note that this implies that two elastic threads in a client cannot share a flow to a server.

IX does have a small number of shared structures, including some that require synchronization on updates. For example, the ARP table is shared by all elastic threads and is protected by RCU locks [41]. Hence, the common case reads are coherence-free but the rare updates are not. RCU objects are garbage collected after a quiescent period that spans the time it takes each elastic thread to finish a run to completion cycle.

IX requires synchronization when the control plane reallocates resources between dataplanes. For instance, when a core is revoked from a dataplane, the corresponding network flows must be assigned to another elastic thread. Such events are rare because resource allocation happens in a coarse-grained manner. Finally, the application code may include inter-thread communication and synchronization. While using IX does not eliminate the need to develop scalable application code, it ensures that there are no scaling bottlenecks in the system and protocol processing code.

4.5 Security Model

The IX API and implementation has a cooperative flow control model between application code and the network-processing stack. Unlike user-level stacks, where the application is trusted for correct networking behavior, the IX protection model makes few assumptions about the application. A malicious or misbehaving application can only hurt itself. It cannot corrupt the networking stack or affect other applications. All application code in IX runs in user-mode, while dataplane code runs in protected ring 0. Applications cannot access dataplane memory, except for read-only message buffers. No sequence of batched system calls or other user-level actions can be used to violate correct adherence to TCP and other network specifications. Furthermore, the dataplane can be used to enforce network security policies, such as firewalling and access control lists. The IX security model is as strong as conventional kernel-based networking stacks, a feature that is missing from all recently proposed user-level stacks.

The IX dataplane and the application collaboratively manage memory. To enable zero-copy operation, a buffer used for an incoming packet is passed read-only to the ap-

plication, using virtual memory protection. Applications are encouraged (but not required) to limit the time they hold message buffers, both to improve locality and to reduce fragmentation because of the fixed size of message buffers. In the transmit direction, zero-copy operation requires that the application must not modify outgoing data until reception is acknowledged by the peer, but if the application violates this requirement, it will only result in incorrect data payload.

Since elastic threads in IX execute both the network stack and application code, a long running application can block further network processing for a set of flows. This behavior in no way affects other applications or dataplanes. We use a timeout interrupt to detect elastic threads that spend excessive time in user mode (e.g., in excess of 10ms). We mark such applications as non-responsive and notify the control plane.

The current IX prototype does not yet use an IOMMU. As a result, the IX dataplane is trusted code that has access to descriptor rings with host-physical addresses. This limitation does not affect the security model provided to applications.

5 Evaluation

We compared IX to a baseline running the most recent Linux kernel and to mTCP [29]. Our evaluation uses both networking microbenchmarks and a widely deployed, event-based application. In all cases, we use TCP as the networking protocol.

5.1 Experimental Methodology

Our experimental setup consists of a cluster of 24 clients and one server connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The client machines are a mix of Xeon E5-2637 @ 3.5 Ghz and Xeon E5-2650 @ 2.6 Ghz. The server is a Xeon E5-2665 @ 2.4 Ghz with 256 GB of DRAM. Each client and server socket has 8 cores and 16 hyperthreads. All machines are configured with Intel x520 10GbE NICs (82599EB chipset). We connect clients to the switch through a single NIC port, while for the server it depends on the experiment. For *10GbE* experiments, we use a single NIC port, and for *4x10GbE* experiments, we use four NIC ports bonded by the switch with a L3+L4 hash.

Our baseline configuration in each machine is an Ubuntu LTS 14.0.4 distribution, updated to the 3.16.1 Linux kernel, the most recent at time of writing. We enable hyperthreading when it improves performance. Except for §5.2, client machines always run Linux. All power management features are disabled for all systems in all experiments. Jumbo frames are never enabled. All

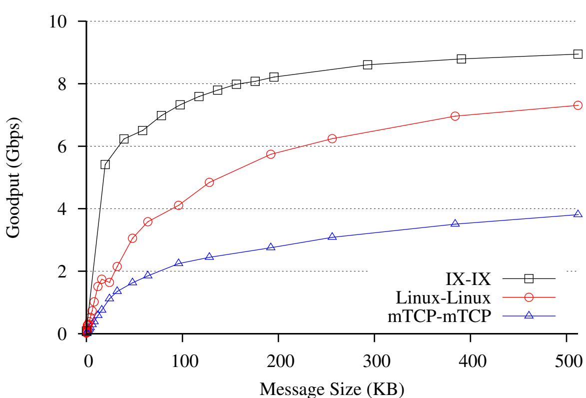


Figure 2: NetPIPE performance for varying message sizes and system software configurations.

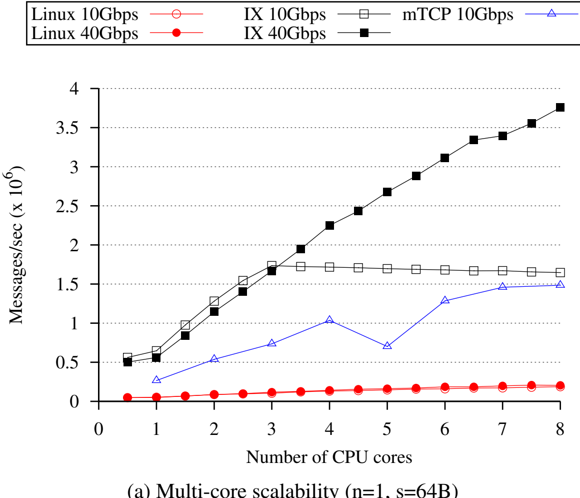
Linux workloads are pinned to hardware threads to avoid scheduling jitter, and background tasks are disabled.

The Linux client and server implementations of our benchmarks use the `libevent` framework with the `epoll` system call. We downloaded and installed mTCP from the public-domain release [30], but had to write the benchmarks ourselves using the mTCP API. We run mTCP with the 2.6.36 Linux kernel, as this is the most recent supported kernel version. We report only 10GbE results for mTCP, as it does not support NIC bonding. For IX, we bound the maximum batch size to $B = 64$ packets per iteration, which maximizes throughput on microbenchmarks (see §6).

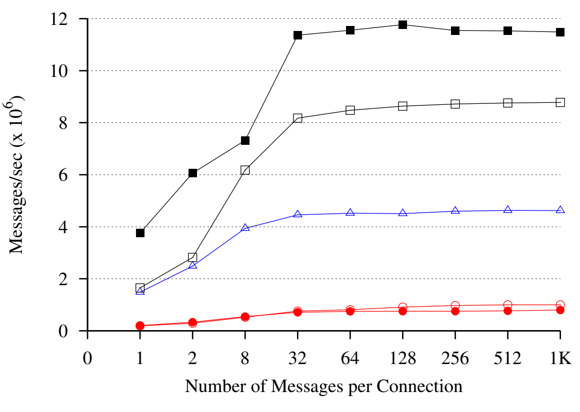
5.2 Latency and Single-flow Bandwidth

We first evaluated the latency of IX using NetPIPE, a popular ping-pong benchmark, using our 10GbE setup. NetPIPE simply exchanges a fixed-size message between two servers and helps calibrate the latency and bandwidth of a single flow [57]. In all cases, we run the same system on both ends (Linux, mTCP, or IX).

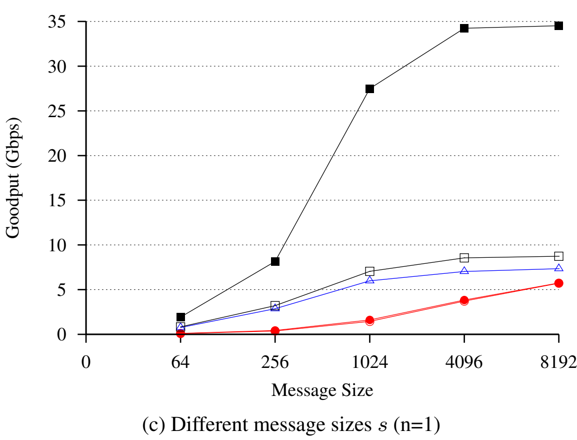
Fig. 2 shows the goodput achieved for different message sizes. Two IX servers have a one-way latency of 5.7µs for 64B messages and achieve goodput of 5Gbps, half of the maximum, with messages as small as 20KB. In contrast, two Linux servers have a one-way latency of 24µs and require 385KB messages to achieve 5Gbps. The differences in system architecture explain the disparity: IX has a dataplane model that polls queues and processes packets to completion whereas Linux has an interrupt model, which wakes up the blocked process. mTCP uses aggressive batching to offset the cost of context switching [29], which comes at the expense of higher latency than both IX and Linux in this particular test.



(a) Multi-core scalability (n=1, s=64B)



(b) n round-trips per connection. (s=64B)



(c) Different message sizes s (n=1)

Figure 3: Multi-core scalability and high connection churn for 10GbE and 4x10GbE setups. In (a), half steps indicate hyperthreads.

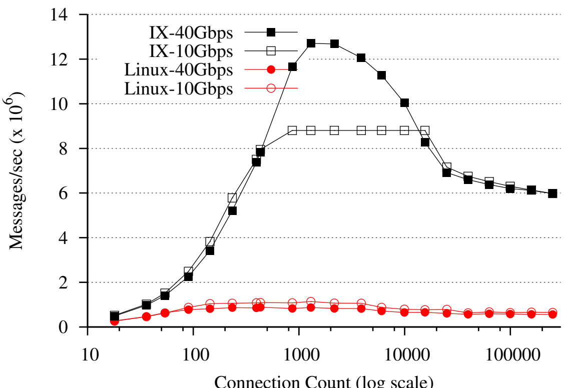


Figure 4: Connection scalability for the 10GbE and 4x10GbE configurations.

5.3 Throughput and Scalability

We evaluate IX's throughput and multi-core scalability with the same benchmark used to evaluate MegaPipe [24] and mTCP [29]. 18 clients connect to a single server listening on a single port, send a remote request of size s bytes, and wait for an echo of a message of the same size. Similar to the NetPIPE benchmark, while receiving the message, the server holds off its echo response until the message has been entirely received. Each client performs this synchronous remote procedure call n times before closing the connection. As in [29], clients close the connection using a reset (`TCP RST`) to avoid exhausting ephemeral ports.

Fig. 3 shows the message rate or goodput for both the 10GbE and the 40GbE configurations as we vary the number of cores used, the number of round-trip messages per connection, and the message size respectively. For the 10GbE configuration, the results for Linux and mTCP are consistent with those published in the mTCP paper [29]. For all three tests (core scaling, message count scaling, message size scaling), IX scales more aggressively than mTCP and Linux. Fig. 3a shows that IX needs only 3 cores to saturate the 10GbE link whereas mTCP requires all 8 cores. On Fig. 3b for 1024 round-trips per connection, IX delivers 8.8 million messages per second, which is 1.9× the throughput of mTCP and of and 8.8× that of Linux. With this packet rate, IX achieves line rate and is limited only by 10GbE bandwidth.

Fig. 3 also shows that IX scales well beyond 10GbE to a 4x10GbE configuration. Fig. 3a shows that IX linearly scales to deliver 3.8 million TCP connections per second on 4x10GbE. Fig. 3b shows a speedup of 2.3× with $n = 1$ and of 1.3× with $n = 1024$ over 10GbE IX. Finally, Fig. 3c shows IX can deliver 8KB messages with a goodput of 34.5 Gbps, for a wire throughput of 37.9 Gbps,

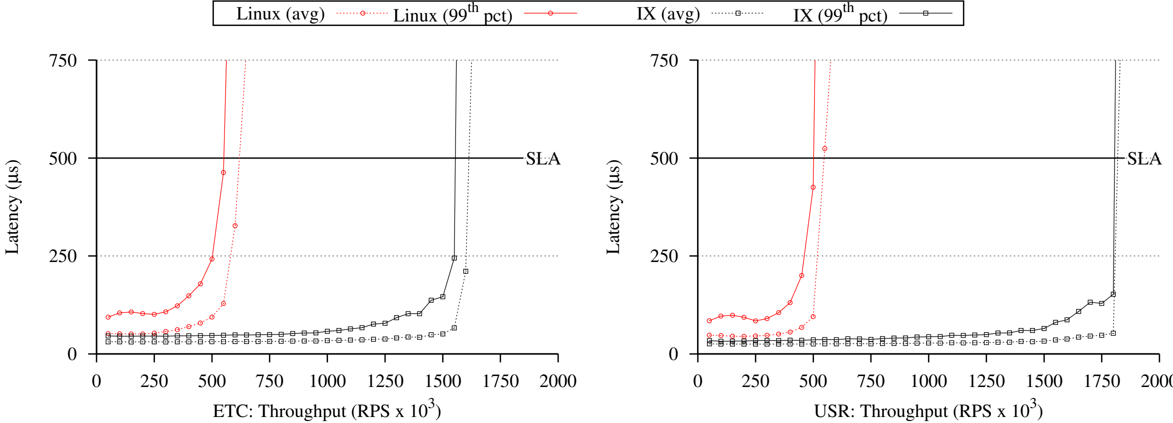


Figure 5: Average and 99th percentile latency as a function of throughput for two memcached workloads.

out of a possible 39.7Gbps. Overall, IX makes it practical to scale protected TCP/IP processing beyond 10GbE, even with a single socket multi-core server.

5.4 Connection Scalability

We also evaluate IX's scalability when handling a large number of concurrent connections on the 4x10GbE setup. 18 client machines runs n threads, with each thread repeatedly performing a 64B remote procedure call to the server with a variable number of active connections. We experimentally set $n = 24$ to maximize throughput. We report the maximal throughput in messages per second for a range of total established connections.

Fig. 4 shows up to 250,000 connections, which is the upper bound we can reach with the available client machines. As expected, throughput increases with the degree of connection concurrency, but then decreases for very large connections counts due to the increasingly high cost of multiplexing among open connections. At the peak, IX performs 10× better than Linux, consistent with the results from Fig. 3b. With 250,000 connections and 4x10GbE, IX is able to deliver 47% of its own peak throughput. We verified that the drop in throughput is not due to an increase in the instruction count, but instead can be attributed to the performance of the memory subsystem. Intel's Data Direct I/O technology, an evolution of DCA [26], eliminates nearly all cache misses associated with DMA transfers when given enough time between polling intervals, resulting in as little as 1.4 L3 cache misses per message for up to 10,000 concurrent connections, a scale where all of IX's data structures fit easily in the L3 cache. In contrast, the workload averages 25 L3 cache misses per message when handling 250,000 concurrent connections. At high connection counts, the working set of this workload is dominated by the TCP connection state and does not fit into the processor's L3 cache. Nevertheless, we believe that further optimizations in the size and access pattern of lwIP's TCP/IP protocol control block structures can substantially reduce this handicap.

5.5 Memcached Performance

Finally, we evaluated the performance benefits of IX with `memcached`, a widely deployed, in-memory, key-value store built on top of the `libevent` framework [42]. It is frequently used as a high-throughput, low-latency caching tier in front of persistent database servers. `memcached` is a network-bound application, with threads spending over 75% of execution time in kernel mode for network processing [36]. It is a difficult application to scale because the common deployments involve high connection counts for `memcached` servers and small-sized requests and replies [2, 46].

We use the `mutilate` load-generator to place a selected load on the server in terms of requests per second (RPS) and measure response latency [35]. `mutilate` coordinates a large number of client threads across multiple machines to generate the desired RPS load, while a separate unloaded client measures latency by issuing one request at the time. We configure `mutilate` to generate load representative of two workloads from Facebook [2]: the ETC workload that represents that highest capacity deployment in Facebook, has 20B–70B keys, 1B–1KB values, and 75% GET requests; and the USR workload that represents deployment with most GET requests in Facebook, has short keys (<20B), 2B values, and 99% GET requests. In USR, almost all traffic involves minimum-sized TCP packets. Each request is issued separately (no `multiget` operations). However, clients are permitted to pipeline up to four requests per connection if needed to keep up with their target request rate. We use 23 client machines to generate load for a total of 1,476 connections to the `memcached` server.

To provide insights into the full range of system behaviors, we report average and 99th percentile latency as a function of the achieved throughput. The 99th percentile latency captures tail latency issues and is the most relevant metric for datacenter applications [14]. Most commercial memcached deployments provision each server so that the 99th percentile latency does not exceed 200µs to 500µs. We carefully tune the Linux baseline setup according to the guidelines in [36]: we pin memcached threads, configure interrupt-distribution based on thread-affinity, and tune interrupt moderation thresholds. We believe that our baseline Linux numbers are as tuned as possible for this hardware using the open-source version of memcached-1.4.18. We report the results for the server configuration that provides the best performance: 8 cores with Linux, but only 6 with IX.

Porting memcached to IX primarily consisted of adapting it to use our event library. In most cases, the port was straightforward, replacing Linux and libevent function calls with their equivalent versions in our API. We did yet not attempt to tune the internal scalability of memcached [20] or to support zero-copy I/O operations.

Fig. 5 shows the throughput-latency curves for the two memcached workloads for Linux and IX, while Table 2 reports the unloaded, round-trip latencies and maximum request rate that meets a service-level agreement, both measured at the 99th percentile. IX noticeably reduces the unloaded latencies to roughly half. Note that we use Linux clients for these experiments; running IX on clients should further reduce latency.

At high request rates, the distribution of CPU time shifts from being $\sim 75\%$ in the Linux kernel to $< 10\%$ in the IX dataplane kernel. This allows IX to increase throughput by 2.8× and 3.6× for ETC and USR respectively at a 500µs tail latency SLA. The improvement for ETC is lower due to the increased lock contention within the application itself, in particular because it has a higher write frequency. Lock contention within application code is also the reason that IX cannot provide throughput improvements with more than 6 cores.

Configuration	Minimum latency @99th pct	RPS for SLA: $< 500\mu s$ @99th pct
ETC-Linux	94µs	550K
ETC-IX	45µs	1550K
USR-Linux	85µs	500K
USR-IX	32µs	1800K

Table 2: Unloaded latency and maximum RPS for a given service-level agreement for the memcache workloads ETC and USR.

6 Discussion

What makes IX fast: The results in §5 show that a networking stack can be implemented in a protected OS kernel and still deliver wire-rate performance for most benchmarks. The tight coupling of the dataplane architecture, using only a minimal amount of batching to amortize transition costs, causes application logic to be scheduled at the right time, which is essential for latency-sensitive workloads. Therefore, the benefits of IX go beyond just minimizing kernel overheads. The lack of intermediate buffers allows for efficient, application-specific implementations of I/O abstractions such the libix event library. The zero-copy approach helps even when the user-level libraries add a level of copying, as it is the case for the libevent compatible interfaces in libix. The extra copy occurs much closer to the actual use, thereby increasing cache locality. Finally, we carefully tuned IX for multi-core scalability, eliminating constructs that introduce synchronization or coherence traffic.

The IX dataplane optimizations — run to completion, adaptive batching, and a zero-copy API — can also be implemented in a user-level networking stack in order to get similar benefits in terms of throughput and latency. While a user-level implementation would eliminate protection domain crossings, it would not lead to significant performance improvements over IX. Protection domain crossings inside VMX non-root mode add only a small amount of extra overhead, on the order of a single L3 cache miss [7]. Moreover, these overheads are quickly amortized at higher packet rates.

Subtleties of adaptive batching: Batching is commonly understood to trade off higher latency at low loads for better throughput at high loads. IX uses adaptive, bounded batching to actually improve on both metrics. Fig. 6 compares the latency vs. throughput on the USR memcached workload of Fig. 5 for different upper bounds B to the batch size. At low load, B does not impact tail latency, as adaptive batching does not delay processing of pending packets. At higher load, larger values of B improve throughput, by 29% between $B = 1$ to $B = 16$. For this workload, $B \geq 16$ maximizes throughput.

While tuning IX performance, we ran into an unexpected hardware limitation that was triggered at high packet rates with small average batch sizes (i.e. before the dataplane was saturated): the high rate of PCIe writes required to post fresh descriptors at every iteration led to performance degradation as we scaled the number of cores. To avoid this bottleneck, we simply coalesced PCIe writes on the receive path so that we replenished at least 32 descriptor entries at a time. Luckily, we did not have to

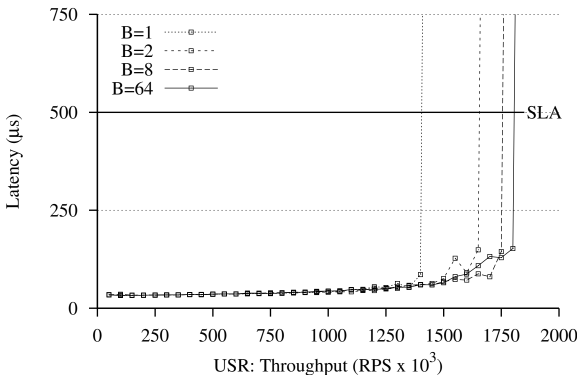


Figure 6: 99th percentile latency as a function of throughput for USR workload from Fig. 5, for different values of the batch bound B.

coalesce PCIe writes on the transmit path, as that would have impacted latency.

Limitations of current prototype: The current IX implementation does not yet exploit IOMMUs or VT-d. Instead, it maps descriptor rings directly into IX memory, using the Linux pagemap interface to determine physical addresses. Although this choice puts some level of trust into the IX dataplane, application code remains securely isolated. In the future, we plan on using IOMMU support to further isolate IX dataplanes. We anticipate overhead will be low because of our use of large pages. Also, the IX prototype currently does not take advantage of the NIC's SR-IOV capabilities, but instead allocates entire physical devices to dataplanes.

We also plan to add support for interrupts to the IX dataplanes. The IX execution model assumes some cooperation from application code running in elastic threads. Specifically, applications should handle events in a quick, non-blocking manner; operations with extended execution times are expected to be delegated to background threads rather than execute within the context of elastic threads. The IX dataplane is designed around polling, with the provision that interrupts can be configured as a fallback optimization to refresh receive descriptor rings when they are nearly full and to refill transmit descriptor rings when they are empty (steps (1) and (6) in Fig 1b). Occasional timer interrupts are also required to ensure full TCP compliance in the event an elastic thread blocks for an extended period.

Future work: This paper focused primarily on the IX dataplane architecture. IX is designed and implemented to support the dynamic addition and removal of elastic threads in order to achieve energy proportional and resource efficient computing. So far we have tested only static configurations. In future work, we will explore control plane issues, including a dynamic runtime that rebalances network flows between available elastic threads in a manner that maintains both throughput and latency constraints.

We will also explore the synergies between IX and networking protocols designed to support microsecond-level latencies and the reduced buffering characteristics of IX deployments, such as DCTCP [1] and ECN [52]. Note that the IX dataplane is not specific to TCP/IP. The same design principles can benefit alternative, potentially application specific, network protocols, as well as high-performance protocols for non-volatile memory access. Finally, we will investigate library support for alternative APIs on top of our low-level interface, such as MegaPipe [24], cooperative threading [65], and rule-based models [60]. Such APIs and programming models will make it easier for applications to benefit from the performance and scalability advantages of IX.

7 Related Work

We organize the discussion topically, while avoiding redundancy with the commentary in §2.3.

Hardware virtualization: Hardware support for virtualization naturally separates control and execution functions, e.g., to build type-2 hypervisors [10, 33], run virtual appliances [55], or provide processes with access to privileged instructions [7]. Similar to IX, Arrakis uses hardware virtualization to separate the I/O dataplane from the control plane [50]. IX differs in that it uses a full Linux kernel as the control plane; provides three-way isolation between the control plane, networking stack, and application; and proposes a dataplane architecture that optimizes for both high throughput and low latency. On the other hand, Arrakis uses Barrelfish as the control plane [6] and includes support for IOMMUs and SR-IOV.

Library operating systems: Exokernels extend the end-to-end principle to resource management by implementing system abstractions via library operating systems linked in with applications [19]. Library operating systems often run as virtual machines [9] used, for instance, to deploy cloud services [39]. IX limits itself to the implementation of the networking stack, allowing applications to implement their own resource management policies, e.g. via the `libevent` compatibility layer.

Asynchronous and zero-copy communication: Systems with asynchronous, batched, or exception-less system calls substantially reduce the overheads associated with frequent kernel transitions and context switches [24, 29, 53, 58]. IX's use of adaptive batching shares similar

benefits but is also suitable for low-latency communication. Zero-copy reduces data movement overheads and simplifies resource management [48]. POSIX OSes have been modified to support zero-copy through page remapping and copy-on-write [11]. By contrast, IX's cooperative memory management enables zero-copy without page remapping. Similar to IX, TinyOS passes pointers to packet buffers between the network stack and the application in a cooperative, zero-copy fashion [37]. However, IX is optimized for datacenter workloads, while TinyOS focuses on memory constrained, sensor environments.

8 Conclusion

We described IX, a dataplane operating system that leverages hardware virtualization to separate and isolate the Linux control plane, the IX dataplane instances that implement in-kernel network processing, and the network-bound applications running on top of them. The IX dataplane provides a native, zero-copy API that explicitly exposes flow control to applications. The dataplane architecture optimizes for both bandwidth and latency by processing bounded batches of packets to completion and by eliminating synchronization on multi-core servers. On microbenchmarks, IX noticeably outperforms both Linux and mTCP in terms of both latency and throughput, scales to hundreds of thousands of active concurrent connections, and can saturate 4x10GbE configurations using a single processor socket. Finally, we show that porting `memcached` to IX removes kernel bottlenecks and improves throughput by up to 3.6×, while reducing tail latency by more than 2×.

Acknowledgements

The authors would like to thank David Mazières for his many insights into the system and his detailed feedback on the paper. We also thank Katerina Argyraki, James Larus, Jacob Leverich, Philip Levis, Willy Zwaenepoel, the anonymous reviewers, and our shepherd Andrew Warfield for their comments. This work was supported by a Google research grant, the Stanford Experimental Datacenter Lab, and the Microsoft-EPFL Joint Research Center. Adam Belay is supported by a VMware Graduate Fellowship.

References

[1] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 2012 ACM SIGMETRICS International conference on Measurement and modeling of computer systems*, pages 53–64, 2012.

[3] L. A. Barroso. Three things that must be done to save the data center of the future (ISSCC 2014 Keynote). `http://www.theregister.co.uk/Print/2014/02/11/google_research_three_things_that_must_be_done_to_save_the_data_center_of_the_future/`, 2014.

[4] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

[5] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA '13)*, pages 237–248, 2013.

[6] A. Baumann, P. Barham, P.-É. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 29–44, 2009.

[7] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI '12)*, pages 335–348, 2012.

[8] S. M. Bellovin. A Look Back at "Security Problems in the TCP/IP Protocol Suite". In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04)*, pages 229–249, 2004.

[9] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997.

[10] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4):12, 2012.

[11] H.-K. J. Chu. Zero-Copy TCP in Solaris. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 253–264, 1996.

[12] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 1–17, 2013.

[13] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 33–48, 2013.

[14] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, 2013.

[15] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, pages 127–144, 2014.

[16] M. Dobrescu, N. Egi, K. J. Argyraki, B.-G. Chun, K. R. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 15–28, 2009.

[17] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[18] A. Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, 1995.

[20] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 371–384, 2013.

[21] M. Fisk and W. Feng. Dynamic Adjustment of TCP Window Sizes. Technical report, Tech. Rep. Los Alamos Unclassified Report (LAUR) 00-3221, Los Alamos National Laboratory, 2000.

[22] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

[23] R. Graham. The C10M Problem. `http://c10m.robertgraham.com`, 2013.

[24] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI '12)*, pages 135–148, 2012.

[25] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 22–22, 2011.

[26] R. Huggahalli, R. R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32st International Symposium on Computer Architecture (ISCA '05)*, pages 50–59, 2005.

[27] Intel Corp. Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/open-source-kernel-enhancements-paper.pdf`, 2013.

[28] Intel Corp. Intel DPDK: Data Plane Development Kit. `http://dpdk.org/`, 2014.

[29] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[30] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. o. Park. mTCP source code release, v. of 2014-02-26. `https://github.com/eunyoung14/mtcp`, 2014.

[31] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *International Conference on Parallel Processing (ICPP '11)*, pages 743–752, 2011.

[32] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *ACM Symposium on Cloud Computing (SOCC '12)*, page 9, 2012.

[33] A. Kivity. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS)*, pages 225–230, July 2007.

[34] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[35] J. Leverich. Mutilate: High-Performance Memcached Load Generator. `https://github.com/leverich/mutilate`, 2014.

[36] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the 9th EuroSys Conference (Eurosys '14)*, page 4, 2014.

[37] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. A. Brewer, and D. E. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 1–14, 2004.

[38] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA '14)*, pages 301–312, 2014.

[39] A. Madhavapeddy, R. Mortier, C. Rotsos, D. J. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 461–472, 2013.

[40] I. Marinos, R. N. M. Watson, and M. Handley. Network Stack Specialization for Performance. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 175–186, 2014.

[41] P. E. McKenney and J. D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[42] memcached – a distributed memory object caching system. `http://memcached.org`, 2014.

[43] Microsoft Corp. Receive Side Scaling. `http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx`, 2014.

[44] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 103–114, 2013.

[45] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[46] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 385–398, 2013.

[47] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 29–41, 2011.

[48] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Trans. Comput. Syst.*, 18(1):37–66, 2000.

[49] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th EuroSys Conference (Eurosys '12)*, pages 337–350, 2012.

[50] S. Peter, J. Li, I. Zhang, D. R. K. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI '14)*, 2014.

[51] N. Provos and N. Mathewson. libevent: an event notification library. `http://libevent.org`, 2003.

[52] K. Ramakrishnan, S. Floyd, D. Black, et al. The Addition of Explicit Congestion Notification (ECN) to IP. IETF RFC 3168, 2001.

[53] L. Rizzo. Revisiting Network I/O APIs: The netmap Framework. *Commun. ACM*, 55(3):45–51, 2012.

[54] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011.

[55] C. P. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the 17th Conference on Systems Administration (LISA '03)*, pages 181–194, 2003.

[56] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th EuroSys Conference (EuroSys '13)*, pages 351–364, 2013.

[57] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, volume 6, 1996.

[58] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI '10)*, pages 33–46, 2010.

[59] Solarflare Communications. Introduction to OpenOnload: Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. `http://www.solarflare.com/content/userfiles/documents/solarflare_openonload_intropaper.pdf`, 2011.

[60] R. Stutsman and J. K. Ousterhout. Toward Common Patterns for Distributed, Concurrent, Fault-Tolerant Code. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS-XIV)*, 2013.

[61] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing Network Protocols at User Level. In *Proceedings of the ACM SIGCOMM 1993 Conference*, pages 64–73, 1993.

[62] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, 2005.

[63] G. Varghese and A. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, pages 25–38, 1987.

[64] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and Effective Fine-Grained TCP Retransmissions for Datacenter Communication. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 303–314, 2009.

[65] J. R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. A. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 268–281, 2003.

[66] WhatsApp, Inc. 1 million is so 2011. `https://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011`, 2012.

[67] D. A. Wheeler. SLOCCount, v2.26. `http://www.dwheeler.com/sloccount/`, 2001.

Willow: A User-Programmable SSD

Sudharsan Seshadri Mark Gahagan Sundaram Bhaskaran Trevor Bunker
Arup De Yanqin Jin Yang Liu Steven Swanson
Computer Science & Engineering, UC San Diego

Abstract

We explore the potential of making programmability a central feature of the SSD interface. Our prototype system, called Willow, allows programmers to augment and extend the semantics of an SSD with application-specific features without compromising file system protections. The *SSD Apps* running on Willow give applications low-latency, high-bandwidth access to the SSD's contents while reducing the load that IO processing places on the host processor. The programming model for SSD Apps provides great flexibility, supports the concurrent execution of multiple SSD Apps in Willow, and supports the execution of trusted code in Willow.

We demonstrate the effectiveness and flexibility of Willow by implementing six SSD Apps and measuring their performance. We find that defining SSD semantics in software is easy and beneficial, and that Willow makes it feasible for a wide range of IO-intensive applications to benefit from a customized SSD interface.

1 Introduction

For decades, computer systems have relied on the same block-based interface to storage devices: reading and writing data from and to fixed-sized sectors. It is no accident that this interface is a perfect fit for hard disks, nor is it an accident that the interface has changed little since its creation. As other system components have gotten faster and more flexible, their interfaces have evolved to become more sophisticated and, in many cases, programmable. However, hard disk performance has remained stubbornly poor, hampering efforts to improve performance by rethinking the storage interface.

The emergence of fast, non-volatile, solid-state memories (such as NAND flash and phase-change memories) has signaled the beginning of the end for painfully slow storage systems, and this demands a fundamental rethinking of the interface between storage software and the storage device. These new memories behave very differently than disks—flash requires out-of-place updates while phase change memories (PCMs) provide byte-addressability—and those differences beg for interfaces that go beyond simple block-based access.

The scope of possible new interfaces is enormously broad and includes both general-purpose and application-specific approaches. Recent work has illustrated some of the possibilities and their potential benefits. For instance, an SSD can support complex atomic operations [10, 32, 35], native caching operations [5, 38], a large, sparse storage address space [16], delegating storage allocation decisions to the SSD [47], and offloading file system permission checks to hardware [8]. These new interfaces allow applications to leverage SSDs' low latency, ample internal bandwidth, and on-board computational resources, and they can lead to huge improvements in performance.

Although these features are useful, the current one-at-a-time approach to implementing them suffers from several limitations. First, adding features is complex and requires access to SSD internals, so only the SSD manufacturer can add them. Second, the code must be trusted, since it can access or destroy any of the data in the SSD. Third, to be cost-effective for manufacturers to develop, market, and maintain, the new features must be useful to many users and/or across many applications. Selecting widely applicable interfaces for complex use cases is very difficult. For example, editable atomic writes [10] were designed to support ARIES-style write-ahead logging, but not all databases take that approach.

To overcome these limitations, we propose to make programmability a central feature of the SSD interface, so ordinary programmers can safely extend their SSDs' functionality. The resulting system, called *Willow*, will allow application, file system, and operating system programmers to install customized (and potentially untrusted) *SSD Apps* that can modify and extend the SSD's behavior.

Applications will be able to exploit this kind of programmability in (at least) four different ways.

- **Data-dependent logic:** Many storage applications perform data-dependent read and write operations to manipulate on-disk data structures. Each data-dependent operation requires a round-trip between a conventional SSD and the host across the system bus (i.e., PCIe, SATA, or SAS) and through the operating system, adding latency and increasing host-side software costs.

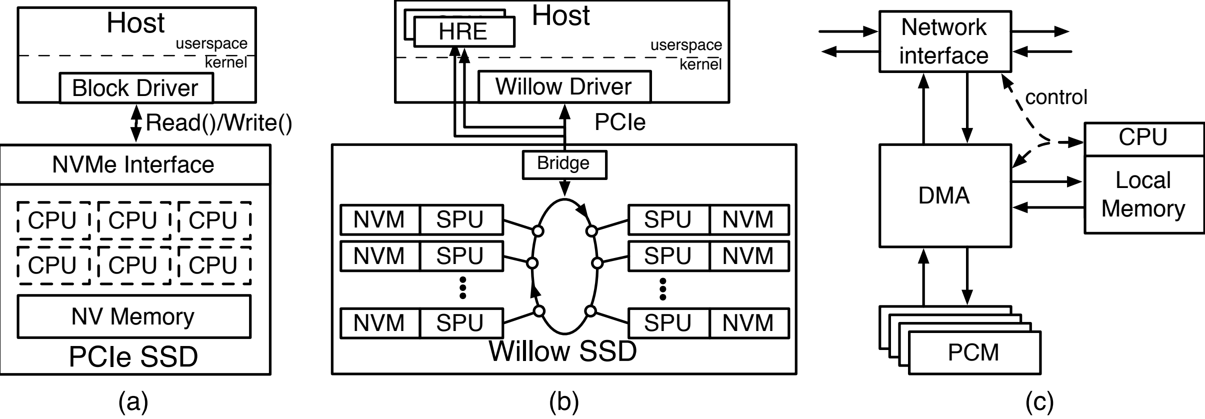


Figure 1: **A conventional SSD vs. Willow.** Although both a conventional SSD (a) and Willow (b) contain programmable components, Willow's computation resources (c) are visible to the programmer and provide a flexible programming model.

- **Semantic extensions:** Storage features like caching and logging require changes to the *semantics* of storage accesses. For instance, a write to a caching device could include setting a dirty bit for the affected blocks.

- **Privileged execution:** Executing privileged code in the SSD will allow it to take over operating and file system functions. Recent work [8] shows that issuing a request to an SSD via an OS-bypass interface is faster than a system call, so running some trusted code in the SSD would improve performance.

- **Data intensive computations:** Moving data-intensive computations to the storage system has many applications, and previous work has explored this direction in disks [37, 1, 19] and SSDs [17, 6, 43] with promising results.

Willow focuses on the first three of these use cases and demonstrates that adding generic programmability to the SSD interface can significantly reduce the cost and complexity of adding new features. We describe a prototype implementation of Willow based on emulated PCM memory that supports a wide range of applications. Then, we describe the motivation behind the design decisions we made in building the prototype. We report on our experience implementing a suite of example SSD Apps. The results show that Willow allows programmers to quickly add new features to an SSD and that applications can realize significant gains by offloading functionality to Willow.

This paper provides an overview of Willow, its programming model, and our prototype in Sections 2 and 3. Section 4 presents and evaluates six SSD Apps, Section 5 places our work in the context of other approaches to integrating programmability into storage devices. Section 6 describes some of the insights we gained from this work, and Section 7 concludes.

2 System Design

Willow revisits the interface that the storage device exposes to the rest of the system, and provides the hardware necessary to support that interface efficiently. This section describes the system from the programmer's perspective, paying particular attention to the programming model and hardware/software interface. Section 3 describes the prototype hardware in more detail.

2.1 Willow system components

Figure 1(a) depicts a conventional storage system with a high-end, PCIe-attached SSD. A host system connects to the SSD via NVM Express (NVMe) [30] over PCIe, and the operating system sends commands and receives responses over that communication channel. The commands are all storage-specific (e.g., read or write a block) and there is a point-to-point connection between the host operating system and the storage device. Modern, high-end SSDs contain several (often many) embedded, programmable processors, but that programmability is not visible to the host system or to applications.

Figure 1(b) shows the corresponding picture of the Willow SSD. Willow's components resemble those in a conventional SSD: it contains several *storage processor units (SPUs)*, each of which includes a microprocessor, an interface to the inter-SPU interconnect, and access to an array of non-volatile memory. Each SPU runs a very small operating system called SPU-OS that manages and enforces security (see Section 2.6 below).

The interface that Willow provides is very different from the interface of a conventional SSD. On the host side, the Willow driver creates and manages a set of ob-

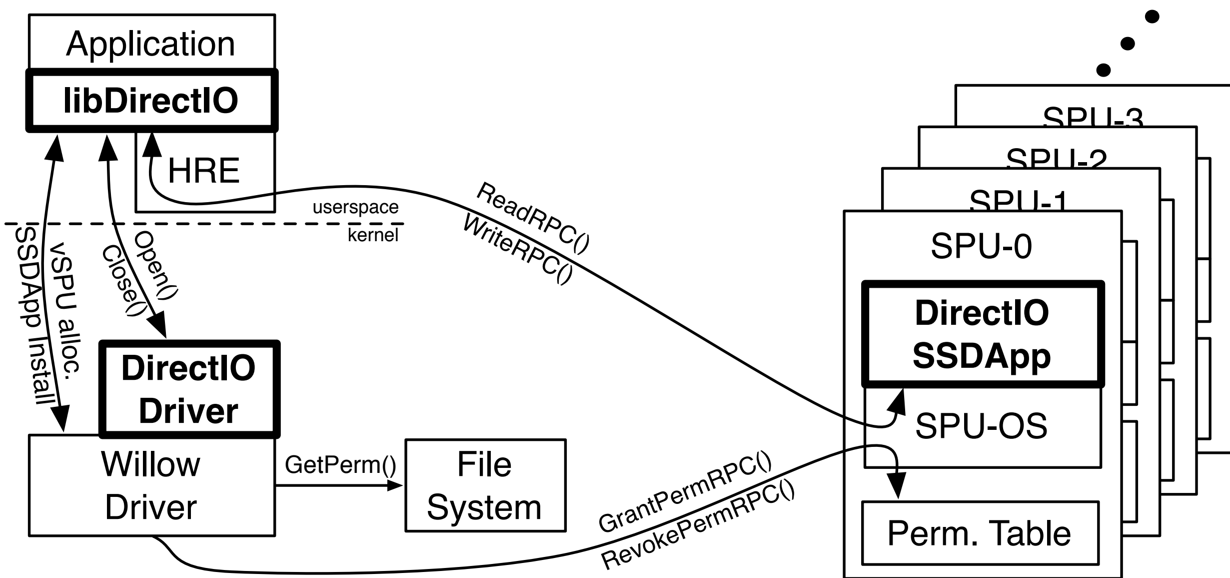


Figure 2: **The Anatomy of an SSD App.** The boldface elements depict three components of an SSD App: a userspace library, the SPU code, and an optional kernel driver. In the typical use case, a conventional file system manages the contents of Willow, and the Willow driver grants access to file extents based on file system permissions.

jects called *Host RPC Endpoints (HREs)* that allow the OS and applications to communicate with SPUs. The HRE is a data structure that the kernel creates and allocates to a process. It provides a unique identifier called the *HRE ID* for sending and receiving RPC requests and lets the process send and receive those requests via DMA transfers between userspace memory and the Willow SSD. The SPUs and HREs communicate over a flexible network using a simple, flexible RPC-based mechanism. The RPC mechanism is generic and does not provide any storage-specific functionality. SPUs can send RPCs to HREs and vice versa.

The final component of Willow is programmable functionality in the form of SSD Apps. Each SSD App consists of three elements: a set of RPC handlers that the Willow kernel driver installs at each SPU on behalf of the application, a library that an application uses to access the SSD App, and a kernel module, if the SSD App requires kernel support. Multiple SSD Apps can be active at the same time.

Below, we describe the high-level system model, the programming model, and the security model for both SPUs and HREs.

2.2 The Willow Usage Model

Willow's design can support many different usage models (e.g., a system could use it as a tightly-coupled network of "wimpy" compute nodes with associated storage). Here, however, we focus on using Willow as a conventional storage device that also provides programmability features. This model is particularly useful because it allows for incremental adoption of Willow's features and ensures that legacy applications can use Willow without modification.

In this model, Willow runs an SSD App called `Base-IO` that provides basic block device functionality (i.e., reading and writing data from and to storage locations). `Base-IO` stripes data across the SPUs (and their associated banks of non-volatile memory) in 8 kB segments. `Base-IO` (and all the other SSD Apps we present in this paper) runs identical code at each SPU. We have found it useful to organize data and computation in this way, but Willow does not require it.

A conventional file system manages the space on Willow and sets permissions that govern access to the data it holds. The file system uses the `Base-IO` block device interface to maintain metadata and provide data access to applications that do not use Willow's programmability.

To exploit Willow's programmability, an application needs to install and use an additional SSD App. Figure 2 illustrates this process for an SSD App called `Direct-IO` that provides an OS-bypass interface that avoids system call and file system overheads for common-case reads and writes (similar to [8]). The figure shows the software components that comprise `Direct-IO` in bold. To use `Direct-IO`, the application uses the `Direct-IO`'s userspace library, `libDirectIO`. The library asks the operating system to install `Direct-IO` in Willow and requests an HRE from the Willow driver to allow it to communicate with the Willow SSD.

`Direct-IO` also includes a kernel module that

libDirectIO invokes when it needs to open a file on behalf of the application. The Direct-IO kernel module asks the Willow driver to grant the application permission to access the file. The driver requests the necessary permission information from the file system and issues trusted RPCs to SPU-OS to install the permission for the file extents the application needs to access in the SPU-OS permission table. Modern file systems already include the ability to query permissions from inside the kernel, so no changes to the file system are necessary.

Base-IO and Direct-IO are "standard equipment" on Willow, since they provide functions that are useful for many other SSD Apps. In particular, other SSD Apps can leverage Direct-IO's functionality to implement arbitrary, untrusted operations on file data.

2.3 Building an SSD App

SSD Apps comprise interacting components running in multiple locations: in the client application (e.g., libDirectIO), in the host-side kernel (e.g., the Direct-IO kernel module), and in the Willow SSD. To minimize complexity, code in all three locations uses a common set of interfaces to implement SSD App functionality. In the host application and the kernel, the HRE library implements these interfaces, while in Willow, SPU-OS implements them. The interfaces provide the following capabilities:

1. **Send an RPC request:** SPUs and HREs can issue RPC requests to SPUs, and SPUs can issue RPCs to HREs. RPC delivery is non-reliable (due to limited buffering at the receiver), and all-or-nothing (i.e., the recipient will not receive a partial message). The sender is notified upon successful (or failed) delivery of the message. Willow supports both synchronous and asynchronous RPCs.
2. **Receive an RPC request:** RPC requests carry an *RPC ID* that specifies which SSD App they target and which handler they should invoke. When an RPC request arrives at an SPU or HRE, the runtime (i.e., the HRE library or SPU-OS) invokes the correct handler for the request.
3. **Send an RPC response:** RPC responses are short, fixed-length messages that include a result code and information about the request it responds to. RPC response delivery is reliable.
4. **Initiate a data transfer:** An RPC handler can asynchronously transfer data between the network interface, local memory, and the local non-volatile memory (for SPUs only).
5. **Allocate local memory:** SSD Apps can declare static variables to allocate space in the SPU's local data memory, but they cannot allocate SPU memory dynamically. Code on the host can allocate data statically or on the heap.
6. **General purpose computation:** SSD Apps are written in C, although the standard libraries are not available on the SPUs.

In addition to these interfaces, the host-side HRE library also provides facilities to request HREs from the Willow driver and install SSD Apps.

This set of interfaces has proved sufficient to implement a wide range of different applications (see Section 4), and we have found them flexible and easy to use. However, as we gain more experience building SSD Apps, we expect that opportunities for optimization, new capabilities, and bug-preventing restrictions on SSD Apps will become apparent.

2.4 The SPU Architecture

In modern SSDs (and in our prototype), the embedded processor that runs the SSD's firmware offers only modest performance and limited local memory capacity compared to the bandwidth that non-volatile memory and the SSD's internal interconnect can deliver.

In addition, concerns about power consumption (which argue for lower clock speeds) and cost (which argue for simple processors) suggest this situation will persist, especially as memory bandwidths continue to grow. These constraints shape both the Willow hardware we propose and the details of the RPC mechanism we provide.

The SPU has four hardware components we use to implement the SSD App toolkit (Figure 1(c)):

1. **SPU processor:** The processor provides modest performance (perhaps 100s of MIPS) and kilobytes of per-SPU instruction and data memory.
2. **Local non-volatile memory:** The array of non-volatile memory can read or write data at over 1 GB/s.
3. **Network interface:** The network provides gigabytes-per-second of bandwidth to match the bandwidth of the local non-volatile memory array and the link bandwidth to the host system.
4. **Programmable DMA controller:** The DMA controller routes data between non-volatile memory, the network port, and the processor's local data memory. It can handle the full bandwidth of the network and local non-volatile memory.

The DMA controller is central to the design of both the SPU and the RPC mechanism, since it allows the modestly powerful processor to handle high-bandwidth streams of data. We describe the RPC interface in the following section.

The SPU runs a simple operating system (SPU-OS) that provides simple multi-threading, works with the Willow host-side driver to manage SPU memory resources, implements protection mechanisms that allow multiple SSD Apps to be active at once, and enforces the

```
void Read_Handler (RPCHdr_t *request_hdr) {        // RPCHdr_t part of the RPC interface
    // Parse the incoming RPC
    BaseIOCmd_t cmd;
    RPCReceiveBytes(&cmd, sizeof(BaseIOCmd_t));    // DMA the IO command header
    RPCResp_t response_hdr;                        // Allocate response
    RPCCreateResponse(request_hdr,                 // populate the response
                      &response_hdr,
                      RPC_SUCCESS);
    RPCSendResponse(response_hdr);                 // Send the response

    // Send the read data back via a second RPC
    CPUID_t dst = request_hdr->src;
    RPCStartRequest(dst,                           // Destination PU
                    sizeof(IOCmd_t) + cmd.length,  // Request body length
                    READ_COMPLETE_HANDLER);        // Read completion RPC ID
    RPCAppendRequest(LOCAL_MEMORY_PORT,            // Source DMA port
                     sizeof(BaseIOCmd_t),          // IO command header size
                     &cmd);                        // IO command header address
    RPCAppendRequest(NV_MEMORY_PORT,               // Source DMA Port
                     cmd.length,                   // Bytes to read
                     cmd.addr);                    // Read address
    RPCFinishRequest();                            // Complete the request
}
```

Figure 3: **READ() implementation for `Base-IO`.** Handling a READ() requires parsing the header on the RPC request and then sending requested data from non-volatile memory back to host via another RPC.

file system's protection policy for non-volatile storage. Section 2.6 describes the protection facilities in more detail.

2.5 The RPC Interface

The RPC mechanism's design reflects the constraints of the hardware described above. Given the modest performance of the SPU processor and its limited local memory, buffering entire RPC messages at the SPU processor is not practical. Instead, the RPC library parses and assembles RPC requests in stages. The code in Figure 3 illustrates how this works for a simplified version of the READ() RPC from `Base-IO`.

When an RPC arrives, SPU-OS copies the RPC header into a local buffer using DMA and passes the buffer to the appropriate handler (`Read_Handler`). That handler uses the DMA controller to transfer the RPC parameters into the SPU processor's local memory (`RPCReceiveBytes`). The header contains generic information (e.g., the source of the RPC request and its size), while the parameters include command-specific values (e.g., the read or write address). The handler uses one or more DMA requests to process the remainder of the request. This can include moving part of the request to the processor's local memory for examination or performing bulk transfers between the network port and the non-volatile memory bank (e.g., to implement a write). In the example, no additional DMA transfers are needed.

The handler sends a fixed-sized response to the RPC request (`RPCCreateResponse` and `RPCSendResponse`). Willow guarantees the reliable delivery of fixed-size responses (acks or nacks) by guaranteeing space to receive them when the RPC is sent. If the SSD App needs to send a response that is longer than 32 bits (e.g., to return the data for a read), it must issue an RPC to the sender. If there is insufficient buffer space at the receiver, the inter-SPU communication network can drop packets. In practice, however, dropped packets are exceedingly rare.

The process of issuing an RPC to return the data follows a similar sequence of steps. The SPU gives the network port the destination and length of the message (`RPCStartRequest`). Then it prepares any headers in local memory and uses the DMA controller to transfer them to the network interface (`RPCAppendRequest`). Further DMA requests can transfer data from non-volatile memory or processor memory to the network interface to complete the request. In this case, the SSD App transfers the read data from the non-volatile memory. Finally, it makes a call to signal the end of the message (`RPCFinishRequest`).

2.6 Protection and sharing in Willow

Willow has several features that make it easy for users to build and deploy useful SSD Apps: Willow supports untrusted SSD Apps, protects against malicious SSD Apps (assuming the host-side kernel is not compromised), allows multiple SSD Apps to be active simultaneously, and allows one SSD App to leverage functionality that another provides. Together these four features allow a user to build and use an SSD App without the permission of a system administrator and to focus on the functionality

specific to his or her particular application.

Providing these features requires a suite of four protection mechanisms. First, it must be clear which host-side process is responsible for the execution of code at the SPU, so SPU-OS can enforce the correct set of protection policies. Second, the SPU must allow an SSD App to access data stored in Willow only if the process that initiated the current RPC has access rights to that data. Third, the SPU must restrict an SSD App to accessing only its own memory and executing only its own code. Finally, it must allow some control transfers between SSD Apps so the user can compose SSD Apps. We address each of these below.

Tracking responsibility: The host system is responsible for setting protection policy for Willow, and it does so by associating permissions with operating system processes. To correctly enforce the operating system's policies, SPU-OS must be able to determine which process is responsible for the RPC handler that is currently running.

To facilitate this, Willow tracks the *originating HRE* for each RPC. An HRE is the originating HRE for any RPCs it makes and for any RPCs that an SPU makes as a result of that RPC and any subsequent RPCs. The PCIe interface hardware in the Willow SSD sets the originating HRE for the initial RPC, and SPU hardware and SPU-OS propagate it within the SSD. As a result, the originating HRE ID is unforgeable and serves as a capability [23].

To reduce cache coherence traffic, it is useful to give each thread in a process its own HRE. The Willow driver allocates HREs so that the high-order bits of the HRE ID are the same for every HRE belonging to a single process.

Non-volatile storage protection: To limit access to data in the non-volatile memory banks, SPU-OS maintains a set of permissions for each process at each SPU. Every time the SSD App uses the DMA controller to move data to or from non-volatile memory, SPU-OS checks that the permissions for the originating HRE (and therefore the originating process) allow it. The worst-case permission check latency is 2 μs.

The host-side kernel driver installs extent-based permission entries on behalf of a process by issuing privileged RPCs to SPU-OS. The SPU stores the permissions for each process as a splay tree to minimize permission check time. Since the SPU-OS permission table is fixed size, it may evict permissions if space runs short. If a request needs an evicted permission entry, a "permission miss" occurs, and the DMA transfer will fail. In response, SPU-OS issues an RPC to the kernel. The kernel forwards the request to the SSD App's kernel module (if it has one), and that kernel module is responsible for resolving the miss. Most of our SSD Apps use the `Direct-IO` kernel module to manage permissions, and it will re-install the permission entry as needed.

Code and Data Protection: To limit access to the code and data in the SPU processor's local memory, the SPU processor provides segment registers and disallows access outside the current segment. Each SSD App has its own data and instruction segments that define the base address and length of the instruction and data memory regions it may access. Accesses outside the SSD App's segment raise an exception and cause SPU-OS to notify the kernel via an RPC, and the kernel, in turn, notifies the applications that the SSD App is no longer available. SPU-OS provides a trusted RPC dispatch mechanism for incoming messages. This mechanism sets the segment registers according to the SSD App that the RPC targets.

The host-side kernel is in charge of managing and statically allocating SPU instruction and data memory to the active SSD Apps. Overlays could extend the effective instruction and data memory size (and are common in commercial SSD controller firmware), but we have not implemented them in our prototype.

Limiting access to RPCs: A combination of hardware and software restricts access to some RPCs. This allows safe composition of SSD Apps and allows SSD Apps to create RPCs that can be issued only from the host-side kernel.

To support composition, SPU-OS provides a mechanism for changing segments as part of a function call from one SSD App to another. An *SSD App-intercall table* in each SPU controls which SSD Apps are allowed to invoke one another and which function calls are allowed. A similar mechanism restricts which RPCs one SSD App can issue to another.

To implement kernel-only RPCs, we use the convention that a zero in the high-order bit of the HRE ID means the HRE belongs to the kernel. RPC implementations can check the ID and return failure when a non-kernel HRE invokes a protected RPC.

SSD Apps can use this mechanism to bootstrap more complex protection schemes as needed. For example, they could require the SSD App's kernel module to grant access to userspace HREs via a kernel-only RPC.

3 The Willow Prototype

We have constructed a prototype Willow SSD that implements all of the functionality described in the previous section. This section provides details about the design.

The prototype has eight SPUs and a total storage capacity of 64 GB. It is implemented using a BEE3 FPGA-based prototyping system [4]. The BEE3 connects to a host system over a PCIe 1.1x8. The link provides 2 GB/s of full-duplex bandwidth.

Each of the four FPGAs that make up a BEE3 hosts

Description	Name	LOC (C)	Devel. Time (Person-months)
Simple IO operations [7]	`Base-IO`	1500	1
Virtualized SSD interface with OS bypass and permission checking [8]	`Direct-IO`	1524	1.2
Atomic writes tailored for scalable database systems based on [10]	`Atomic-Write`	901	1
Direct-access caching device with hardware support for dirty data tracking [5]	`Caching`	728	1
SSD acceleration for MemcacheDB [9]	`Key-Value`	834	1
Offload file appends to the SSD	`Append`	1588	1

Table 1: **SSD Apps.** Implementing and testing each SSD App required no more than five weeks and less than 1600 lines of code.

two SPUs, each attached to an 8 GB bank of DDR2 DRAM. We use the DRAM combined with a customized memory controller to emulate phase change memory with a read latency of 48 ns and a write latency of 150 ns. The memory controller implements start-gap wear-leveling [36].

The SPU processor is a 125 MHz RISC processor with a MIPS-like instruction set. It executes nearly one instruction per cycle, on average. We use the MIPS version of gcc to generate executable code for it. For debugging, it provides a virtual serial port and a rich set of performance counters and status registers to the host. The processor has 32 kB of local data memory and 32 kB of local instruction memory.

The kernel driver statically allocates space in the SPU memory to SSD Apps, which constrains the number and size of SSD Apps that can run at once. SPU-OS maintains a permission table in the local data memory that can hold 768 entries and occupies 20 kB of data memory.

The ring in Willow uses round-robin, token-based arbitration, so only one SPU may be sending a message at any time. To send a message, the SPU's network interface waits for the token to arrive, takes possession of it, and transmits its data. To receive a message, the interface watches the header of messages on the ring to identify messages it should remove from the ring. The ring is 128 bits wide and runs at 250 MHz for a total of 3.7 GB/s of bisection bandwidth.

For communication with the HREs on the host, a bridge connects the ring to the PCIe link. The bridge serves as a hardware proxy for the HREs. For each of the HREs, the bridge maintains an upstream (host-bound) and downstream (Willow-bound) queue. This queue-based interface is similar to the scheme that NVMExpress [30] uses to issue and complete IO requests. The bridge in our prototype Willow supports up to 1024 queue pairs, so it can support 1024 HREs on the host.

The bridge also helps enforce security in Willow. Messages from HREs to SPUs travel over the bridge, and the bridge sets the originating HRE fields on those messages depending on which HRE queue they came in on. Since processes can send messages only via the queues for the HREs they control, processes cannot send forged RPC requests.

4 Case Studies

Willow makes it easy for storage system engineers to improve performance by incorporating new capabilities into a storage device. We have evaluated Willow's effectiveness in this regard by implementing six different SSD Apps and comparing their performance to implementations that use a conventional storage interface.

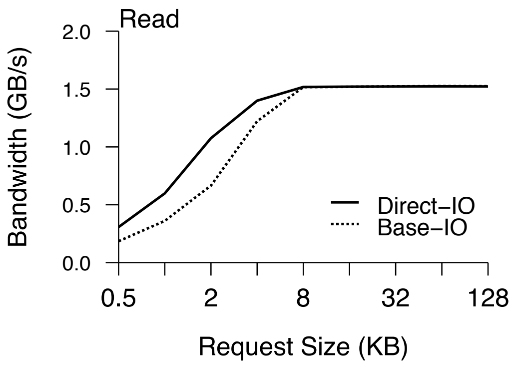
The six applications are: basic IO, IO with OS bypass, atomic-writes, caching, a key-value store, and appending data to a file in the Ext4 filesystem. Table 1 briefly describes all six apps and provides some statistics about their implementations. We discuss each in detail below.

4.1 Basic IO

The first SSD App is `Base-IO`, the SSD App we described briefly in Section 2 that provides basic SSD functionality: READ(), WRITE(), and a few utility operations (e.g., querying the size of the device) that the operating system requires to recognize Willow as a block device.

A Willow SSD with `Base-IO` approximates a conventional SSD, since the SSD's firmware would implement the same functions that `Base-IO` provides. We compare to `Base-IO` throughout this section to understand the performance impact of Willow's programmability features.

Figure 4 plots the performance of `Base-IO`. We collected the data by running XDD [46] on top of XFS. `Base-IO` is able to utilize 78% and 73% of the PCIe bandwidth for read and write, respectively, and can sustain up to 388K read IOPs for small accesses. This level of PCIe utilization is comparable to what we have seen in commercial high-end PCIe SSDs.



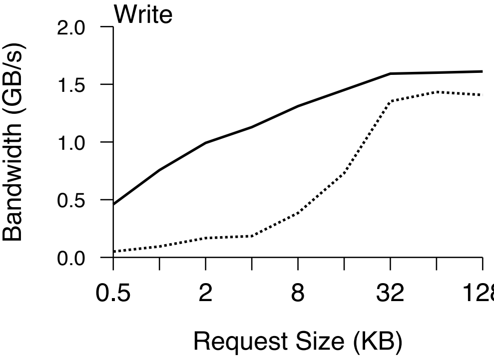


Figure 4: **Bandwidth comparison between `Direct-IO` and `Base-IO`.** Bypassing the kernel with a virtualized SSD interface and software permission checks improves the performance by up to 66% for reads, and 8× for writes, relative to Base-IO.

4.2 Direct-IO

The second SSD App is `Direct-IO`, the OS-bypass interface that allows applications to perform READ() and WRITE() operations without operating system intervention. We described `Direct-IO` in Section 2. `Direct-IO` is similar to the work in [8] and, like that work, `Direct-IO` relies on a userspace library to implement a POSIX-compliant interface for applications.

Figure 4 compares the performance of `Direct-IO` and `Base-IO` running under XFS. `Direct-IO` outperforms `Base-IO` by up to 66% for small reads and 8× for small writes by avoiding system call and file system overheads. The performance gain for writes is larger than for reads because writes require one RPC round trip while reads require two: an RPC from the host to the SSD to send the request and an RPC from the SSD to the host to return the data. `Direct-IO` reduces the cost of the first RPC, but not the second.

Figure 5 breaks down the read latency for 4 kB accesses on three different configurations. All of them share the same hardware (DMA and NVM access) and host-side (command issue, memory copy and software) latencies, but `Direct-IO` saves almost 35% of access latency by avoiding the operating system. The final bar (based on projections) shows that running the SPU at 1 GHz would almost eliminate the impact of SPU software overheads on overall latency, although it would increase power consumption. Such a processor would be easy to implement in a custom silicon version of Willow.

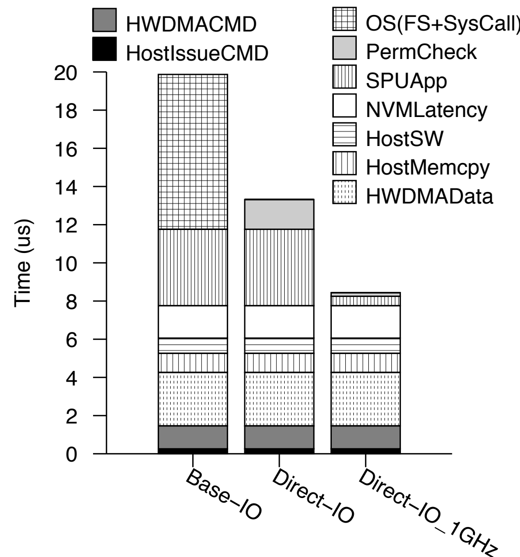


Figure 5: **Read Latency Breakdown.** The bars give the component latencies for `Base-IO` with a file system, for `Direct-IO` on the current SPU processor, and for `Direct-IO` on a hypothetical version of Willow with a 1 GHz processor.

4.3 Atomic Writes

Many storage applications (e.g., file systems and databases) use write-ahead logging (WAL) to enforce strict consistency guarantees on persistent data structures. WAL schemes range from relatively simple journaling mechanisms for file system metadata to the complex ARIES scheme for implementing scalable transactions in databases [27]. Recently, researchers and industry have developed several SSDs with built-in support for multi-part atomic writes [32, 35], including a scheme called MARS [10] that aims to replace ARIES in databases.

MARS relies on a WAL primitive called editable atomic writes (EAW). EAW provides the application with detailed control over where logging information resides inside the SSD and allows it to edit log records prior to committing the atomic operations.

We have implemented EAWs as an SSD App called `Atomic-Writes`. `Atomic-Writes` implements four RPCs—LOGWRITE(), COMMIT(), LOGWRITECOMMIT(), and ABORT()—summarized in Table 2. `Atomic-Writes` makes use of the `Direct-IO` functionality as well.

RPC	Description
LOGWRITE()	Start a new atomic operation and/or add a write to an existing atomic operation.
COMMIT()	Commit an atomic operation.
LOGWRITECOMMIT()	Create and commit an atomic operation comprised of single write.
ABORT()	Abort an atomic operation.

Table 2: **RPCs for `Atomic-Writes`.** The `Atomic-Write` SSD App allows applications to combine multiple writes into a single atomic operation and commit or abort them.

The implementations of LOGWRITE() and COMMIT() illustrate the flexible programmability of Willow's RPC interface. Each SPU maintains the redo-log as a complex persistent data structure for each active transaction. An array of log metadata entries resides in a reserved area of non-volatile memory with each entry pointing to a log record, the data to be written, and the location where it should be written. LOGWRITE() appends an entry to this array and initializes it to add the new entry to the log.

COMMIT() uses a two-phase commit protocol among the SPUs to achieve atomicity. The host library tracks which SPUs are participating in the transaction and selects one of them as the coordinator. In Phase 1, the coordinator broadcasts a "prepare" request to all the SPUs participating in this transaction (including itself). Each participant decides whether to commit or abort and reports back to the coordinator. In Phase 2, if any participant decides to abort, the coordinator instructs all participants to abort. Otherwise the coordinator broadcasts a "commit" request so that each participant plays its local portion of the log and notifies the coordinator when it finishes.

We have modified the Shore-MT [40] storage manager to use MARS and EAW to implement transaction processing. We also fine-tuned EAWs to match how Shore-MT manages transactions, something that would not be possible in the "black box," one-size-fits-all implementation of EAWs that a non-programmable SSD might include. Figure 6 shows the performance difference between MARS and ARIES for TPC-B [44]. MARS scales better than ARIES when increasing thread count and outperforms ARIES by up to 1.5×. These gains are ultimately due to the rich semantics that `Atomic-Writes` provides.

4.4 Caching

SSDs are more expensive and less dense than disks. A cost-effective option for integrating them into storage

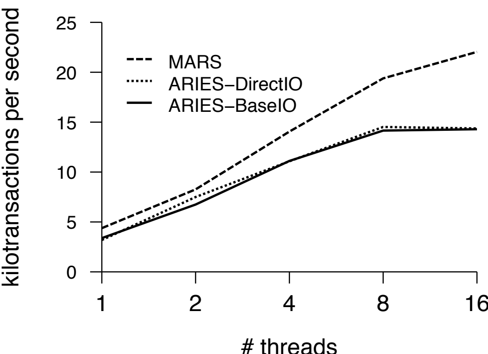


Figure 6: **TPC-B Throughput.** MARS using `Atomic-Writes` yields up to 1.5× throughput gain compared to ARIES using `Base-IO` and `Direct-IO`.

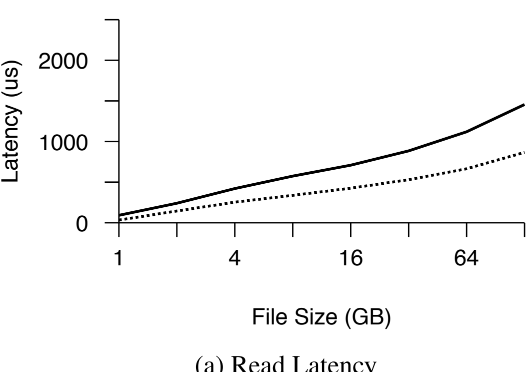
systems is to utilize high performance SSDs as caches for larger conventional backing stores. Traditional SSD caching systems such as FlashCache [41] and bcache [3] implement cache look-up and management operations as a software component in the operating system. Several groups [5, 38] have proposed adding caching-specific interfaces to SSDs in order to improve the performance of the storage system.

We have implemented an SSD App called `Caching` that turns Willow into a caching SSD. `Caching` tracks which data in the cache are dirty, provides support for recovery after failures, and tracks statistics about which data is "hot" to guide replacement policies. It services cache hits directly from user space using `Direct-IO`'s OS-bypass interface. For misses, `Caching` invokes a kernel-based cache manager. Its design is based on Bankshot [5].

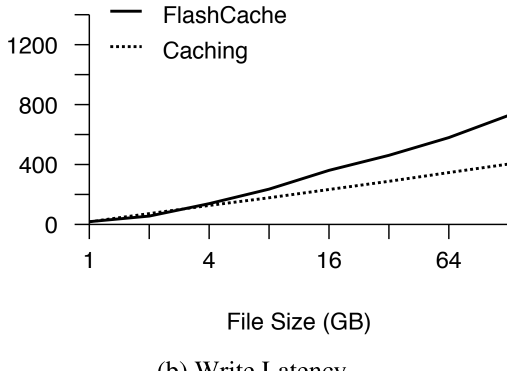
`Caching` transforms Willow into a specialized caching SSD rather than providing application-specific features on top of normal cache access. Instead of using the file system's extent-based protection policy, `Caching` uses a specialized permission mechanism based on large, fixed-size cache chunks (or groups of blocks) that make more efficient use of the SPU's limited local memory. `Caching`'s kernel module uses a privileged kernel-only RPC to install the specialized permission entries and to manage the cache's contents.

To measure `Caching`'s performance we use the Flexible IO Tester (Fio) [14]. We configure Fio to generate Zipf-distributed [2] accesses such that 90% of accesses are to 10% of the data. We vary the file size from 1 GB to 128 GB. We use a 1 GB cache and report average latency after the cache is warm. The backing store is a hard disk.

Figure 7 shows the average read and write latency for 4 kB accesses to FlashCache and `Caching`. Because it is a kernel module, FlashCache uses the `Base-IO` rather than `Direct-IO`. `Caching`'s fully associative



(a) Read Latency



(b) Write Latency

Figure 7: **Latency vs. working set size.** `Caching` offers improved average latency for 4 kB reads (a) and writes (b) compared to FlashCache. As the file sizes grow beyond the cache size of 1 GB, latency approaches that of the backing disk.

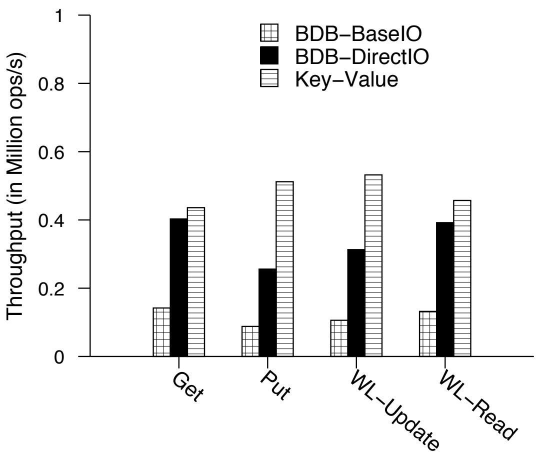


Figure 8: **MemcacheDB performance.** `Key-Value` improves performance of GET() and PUT() operations by 8% and 2× respectively, compared to Berkeley DB running on `Direct-IO`. It improves performance by the same operation by 2× and 4.8× respectively compared to Berkeley DB running on `Base-IO`.

allocation policy allows for more efficient use of cache space, and its ability to allow direct user space access reduces software overheads. `Caching` reduces the miss rate by 6%-23% and improves the cache hit latency for write by 61.8% and read by 36.3%. Combined, these improve read latency by between 1.7 and 2.8× and writes by up to 1.8×.

4.5 Key-Value Store

Key-value stores have proved a very useful tool in implementing a wide range of applications, from smart phone apps to large scale cloud services. Persistent key-value stores such as BerkeleyDB [31], Cassandra [21], and MongoDB [34] rely on complex in-storage data structures (e.g., BTrees or hash tables) to store their data. Traversing those data structures using conventional IO operations results in multiple dependent accesses that consume host CPU cycles and require multiple crossings of the system interconnect (i.e., PCIe). Offloading those dependent accesses to the SSD eliminates much of that latency.

We implement support for key-value operations in an SSD App called `Key-Value`. It provides three RPC functions: PUT() to insert or update a key-value pair, GET() to retrieve the value corresponding to a key, and DELETE() to remove a key-value pair. `Key-Value` stores pairs in a hash table using open chaining to avoid collisions.

`Key-Value` computes the hash of the key on the host and uses the hash value to distribute hash buckets across the SPUs in Willow. For calls to GET() and DELETE(), it passes the hash value and the key (so the SPU can detect matches). For PUT(), it includes the value in addition to the key. All three RPC calls operate on an array of buckets, each containing a linked list of key-value pairs with matching hashes. The SPU code traverses the linked list with a sequence of short DMA requests.

We used MemcacheDB [9] to evaluate `Key-Value`. MemcacheDB [9] combines memcached [26], the popular distributed key-value store, with BerkeleyDB [31], to build a persistent key-value store. MemcacheDB has a client-server architecture, and for this experiment we run it on a single computer that acts both as client (using a 16 thread configuration) and server.

We compare three configurations of MemcacheDB. The first two configurations use BerkeleyDB [31] running on top of `Base-IO` and `Direct-IO` separately to store the key-value pairs. The third replaces BerkeleyDB with a `Key-Value`-based implementation.

We evaluate the performance of GET() and PUT() operations and then measure the overall performance for both update-heavy (50% PUT() / 50% GET()) and read-heavy (5% PUT() / 95% GET()) workloads. Both workloads use random 16-byte keys and 1024-byte values.

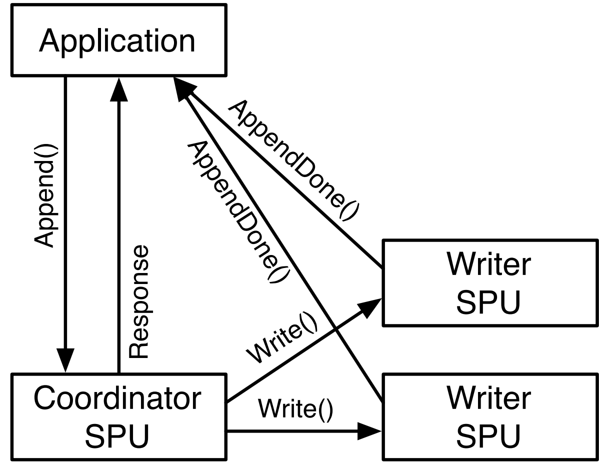


Figure 9: **RPCs to implement APPEND().** The coordinator SPU delegates writing the appended data to the SPUs that host the affected memory banks. Those SPUs notify the host on completion.

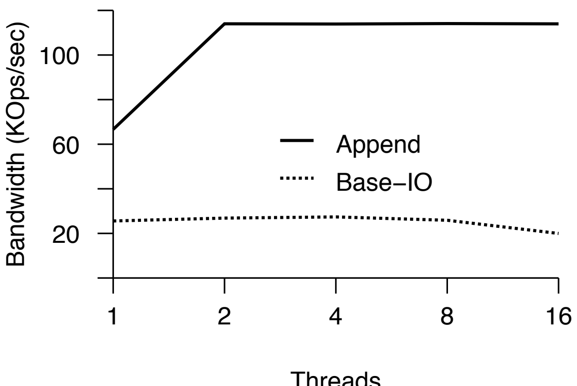


Figure 10: **File append in Willow.** `Append` provides better performance than relying on the operating system to append data to a file.

Figure 8 shows the performance comparison between different MemcacheDB implementations. For GET() and PUT() operations `Key-Value` outperforms the `Direct-IO` configuration by 8.2% and 100% respectively, and improves over the `Base-IO` configuration by 2× and 4.8×. Results for the update- and read-heavy workloads show a similar trend, with `Key-Value` improving performance by between 17% and 70% over the `Direct-IO` configuration and between 2.5× and 4× over the `Base-IO` configuration.

4.6 File system offload

File systems present several opportunities for offloading functionality to Willow to improve performance. We have created an SSD App called `Append` that exploits one of these opportunities, allowing `Direct-IO` to append data to a file (and update the appropriate metadata) from userspace.

`Direct-IO` reduces overheads for most read and write operations by allowing them to bypass the operating system, but it cannot do the same for append operations, since appends require updates to file system metadata. We can extend the OS bypass interface to include appends by building a trusted SSD App that can coordinate with the file system to maintain the correct file length.

`Append` builds upon `Direct-IO` (and `libDirectIO`) and works with a modified version of the Ext4 file system to manage file lengths. The first time an application tries to append to a file, it asks the file system to delegate control of the file's length to `Append`. In response, the file system uses a trusted RPC to tell `Append` where the last extent in the file resides. The file system also sets a flag in the inode to indicate that `Append` has ownership of the file's *logical* length. Since the file system allocates space in 4 kB blocks and may pre-allocate space for the file, the *physical* length of the file is often much longer than the logical length. The physical length remains under the file system's control.

After that, the application can send APPEND() RPCs directly to Willow. Figure 9 illustrates the sequence of RPCs involved. The APPEND() RPCs include the file's inode number and the data to append. The application sends the RPC to the SPU whose ID is the inode number modulo the number of SPUs in Willow.

When the SPU receives an APPEND() RPC, it checks to see whether the application has permissions to append to the file and whether appended data will fit in the physical length of the file. If the permission exists and the data will fit, `Append` issues a special WRITE() to the SPUs that manage the memory that is the target of the append (there may be more than one depending on the size and alignment of the update). While the writes are underway, APPEND() logs the updated length to persistent storage (for crash recovery), and sends a response to the application.

This response does not signal the completion of the APPEND(). Instead, it contains the number of WRITE()s that the coordinating SPU issued and the starting address of the append operation. The WRITE()s for the append notify the host-side application (rather than the coordinating SPU) when they are complete via an APPENDDONE() RPC. When the application has received all of the APPENDDONE() RPCs, it knows the APPEND() is complete. If any of the writes fail, the application needs to re-issue the write using `Direct-IO`.

If the append data will not fit in the physical length of

the file, `Append` sends an "insufficient space" response to the host-side application. The host-side application then invokes the file system to allocate physical space for the file and notify the SPU.

If the file system needs to access the file's length, it retrieves it from the SSD and updates its in-memory data structures.

Figure 10 compares the performance of file appends using `Append` and using `Base-IO`. For `Base-IO` we open the file with O_DSYNC, which provides the same durability guarantees as `Append`. The appends are 1 kB. We modify Ext4 to pre-allocate 64 MB of physical extents. `Append` improves append latency by 2.5× and bandwidth by between 4× and 5.7× with multiple threads.

5 Related Work

Many projects (and some commercial products) have integrated compute capabilities into storage devices, but most of them focus on offloading bulk computation to an active hard drive or (more recently) an SSD.

In the 1970s and 1980s, many advocates of specialized database machines pressed for custom hardware, including processor-per-track or processor-per-head hard disks to achieve processing at storage device. None of these approaches turned out to be successful due to high design complexity and manufacturing cost.

Several systems, including CASSM [42], RAP [33], and RARES [24] provided a processor for each disk track. However, the extra logic required to enable processing ability on each track limited storage density, drove up costs and prevented processor-per-track from finding wide use.

Processor-per-head techniques followed, with the goal of reducing costs by associating processing logic with each read/write head of a moving head hard disk. The Ohio State Data Base Computer (DBC)[18] and SURE [22] each took this approach. These systems demonstrated good performance for simple search tasks, but could not handle more complex computation such as joins or aggregation.

Two different projects, each named Active Disks, continued the trend toward fewer processors, providing just one CPU per disk. The first [37] focused on multimedia, database, and other scan-based operations, and their analysis mainly addressed performance considerations. The second [1] provided a more complete system architecture but supported only stream-based computations called disklets.

Several systems [39, 11] targeted (or have been applied to) databases with programmable in-storage processing resources and some integrated FPGAs [28, 29]. IDisk [19] focused on decision support databases and considered several different software organizations, ranging from running a full-fledged database on each disk to just executing data-intensive kernels (e.g., scans and joins). Willow resembles the more general-purpose programming models for IDisks.

Recently researchers have extended these ideas to SSDs [13, 20], and several groups have proposed offloading bulk computation to SSDs. The work in [17] implements Map-Reduce [12]-style computations in an SSD, and two groups [6, 43] have proposed offloading data analysis for HPC applications to the SSD's processor. Samsung is shipping an SSD with a key-value interface.

Projects that place general computation power into other hardware components, such as programmable NICs, have also been proposed [15, 45, 25]. These devices allow for application-specific code to be placed within the NIC in order to offload network-related computation. This in turn reduces the load of the host OS and CPU in a similar manner to Willow.

Most of these projects focus on bulk computation, and we see that as a reasonable use case for Willow as well, although it would require a faster processor. However, Willow goes beyond bulk processing to include modifying the semantics of the device and allowing programmers to implement complex, control-intensive operations in the SSD itself. Some programmable NICs have taken this approach. Many projects [10, 32, 35, 5, 38, 16, 47, 8, 9] have shown that moving these operations to the SSD is valuable, and making the SSD programmable will open up many new opportunities for performance improvement for both application and operating system code.

6 Discussion

Willow's goal is to expose programmability as a first-class feature of the SSD interface and to make it easier to add new, application-specific functionality to a storage device. Our six example SSD Apps demonstrate that Willow is flexible enough to implement a wide range of SSD Apps, and our experience programming Willow demonstrates that building, debugging, and refining SSD Apps is relatively easy.

`Atomic-Writes` serves as a useful case study in this regard. During its development we noticed that our Willow-aware version of ShoreMT was issuing transactions that comprised several small updates in quick succession. The overhead for sending these LOGWRITE() RPCs was hurting performance. To reduce this overhead, we implemented a new RPC, VECTORLOGWRITE(), that sent multiple IO requests to Willow in a single RPC. Adding this new operation to match ShoreMT's needs took only a couple of days.

Several aspects of Willow's design proved especially helpful. Providing a uniform, generic, and simple programming interface for both HREs and SPUs made Wil-

low easier to use and implement. The RPC mechanism is generic and familiar enough to let us implement most applications in an intuitive way. The simplicity meant that SPU-OS could be both compact and efficient, critical advantages in the Willow SSD's performance- and memory-constrained environment.

SSD Apps' composability was also useful. First, reusing code allowed Willow to make more efficient use of the the available instruction memory. Second, it made developing SSD Apps easier. For instance, most of our SSD App relied on `Direct-IO` to manage basic file access and permissions. Even better, doing so frees the developer from needing to write a custom kernel module and convincing the system administrator to install it.

Willow has the flexibility to implement a wide range of SSD Apps, and the architecture of the Willow SSD provides scalable capacity and supports a great deal of parallelism. However, some trade-offs made in the design present challenges for SSD App developers. We discuss several of these below.

First, striping memory across SPUs provides scalable memory bandwidth, but it also makes it more difficult to implement RPCs that need to make changes across multiple memory banks. The `Append` would have been much simpler if the coordinating SPU had been able to directly access all the file's data.

Second, the instruction memory available at each SPU limits the complexity of SSD Apps, the number of SSD Apps that can execute simultaneously, and the number of permission entries that can reside in the Willow SSD at once. While moving to a custom silicon-based (rather than FPGA-based) controller would help, these resource restrictions would likely remain stringent.

Third, the bandwidth of Willow SSD's ring-based interconnect is much lower than the aggregate bandwidth of the memory banks at the SPUs. This is not a problem for applications that make large transfers mostly between the host and the SSD, since the ring bandwidth is higher than the PCIe link bandwidth. However, it would limit the performance of applications that require large, simultaneous transfers between SPUs.

7 Conclusion

Solid state storage technologies offer dramatic increases in flexibility compared to conventional disk-based storage, and the interface that we use to communicate with storage needs to be equally flexible. Willow offers programmers the ability to implement customized SSD features to support particular applications. The programming interface is simple and general enough to enable a wide range of SSD Apps that can improve performance on a wide range of applications.

Acknowledgements

We would like to thank the reviewers, and especially Ed Nightingale, our shepherd, for their helpful suggestions. We also owe a debt of gratitude to Isabella Furth for her excellent copyediting skills. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. It was also supported in part by NSF award 1219125.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM.

[2] L. A. Adamic and B. A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3:143–150, 2002.

[3] Bcache. http://bcache.evilpiepirate.org/.

[4] http://www.beecube.com/platform.html.

[5] M. S. Bhaskaran, J. Xu, and S. Swanson. BankShot: Caching slow storage in fast non-volatile memory. In *First Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, INFLOW '13, 2013.

[6] S. Boboila, Y. Kim, S. Vazhkudai, P. Desnoyers, and G. Shipman. Active flash: Out-of-core data analytics on flash storage. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12, 2012.

[7] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[8] A. M. Caulfield, T. I. Mollov, L. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, March 2012. ACM.

[9] S. Chu. Memcachedb. http://memcachedb.org/.

[10] J. Coburn, T. Bunker, M. Shwarz, R. K. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation solid-state drives. In *Proceedings of the 24th International Symposium on Operating Systems Principles (SOSP)*, 2013.

[11] G. P. Copeland, Jr., G. J. Lipovski, and S. Y. Su. The architecture of CASSM: A cellular system for non-numeric processing. In *Proceedings of the First Annual Symposium on Computer Architecture*, ISCA '73, pages 121–128, New York, NY, USA, 1973. ACM.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[13] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the 2013 ACM SIG-*

MOD International Conference on Management of Data, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[14] Flexible I/O Tester. http://freecode.com/projects/fio.

[15] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. Spine: A safe programmable and integrated network environment. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, EW 8, pages 7–12, New York, NY, USA, 1998. ACM.

[16] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, Sept. 2010.

[17] Y. Kang, Y. Kee, E. Miller, and C. Park. Enabling cost-effective data processing with smart SSD. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–12, 2013.

[18] K. Kannan. The design of a mass memory for a database computer. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, ISCA '78, pages 44–51, New York, NY, USA, 1978. ACM.

[19] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, Sept. 1998.

[20] S. Kim, H. Oh, C. Park, S. Cho, and S.-W. Lee. Fast, energy efficient scan inside flash memory SSDs. In *Proceedings of ADMS 2011*, 2011.

[21] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[22] H.-O. Leilich, G. Stiege, and H. C. Zeidler. A search processor for data base management systems. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4*, VLDB '78, pages 280–287. VLDB Endowment, 1978.

[23] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[24] C. S. Lin, D. C. P. Smith, and J. M. Smith. The design of a rotating associative memory for relational database applications. *ACM Trans. Database Syst.*, 1(1):53–65, Mar. 1976.

[25] A. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience in offloading protocol processing to a programmable NIC. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 67–74, 2002.

[26] http://memcached.org/.

[27] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[28] R. Mueller and J. Teubner. FPGA: What's in it for a database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 999–1004, New York, NY, USA, 2009. ACM.

[29] R. Mueller, J. Teubner, and G. Alonso. Data processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[30] NVMHCI Work Group. NVM Express. http://nvmexpress.org.

[31] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB.

[32] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 301–311, Washington, DC, USA, 2011. IEEE Computer Society.

[33] E. A. Ozkarahan, S. A. Schuster, and K. C. Sevcik. Performance evaluation of a relational associative processor. *ACM Trans. Database Syst.*, 2(2):175–195, June 1977.

[34] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkeley, CA, USA, 1st edition, 2010.

[35] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 147–160, Berkeley, CA, USA, 2008. USENIX Association.

[36] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[37] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, June 2001.

[38] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[39] S. Schuster, H. B. Nguyen, E. Ozkarahan, and K. Smith. RAP: An associative processor for databases and its applications. *Computers, IEEE Transactions on*, C-28(6):446–458, 1979.

[40] Shore-MT. http://research.cs.wisc.edu/shore-mt/.

[41] M. Srinivasan. FlashCache: A Write Back Block Cache for Linux. https://github.com/facebook/flashcache.

[42] S. Y. W. Su and G. J. Lipovski. CASSM: A cellular system for very large data bases. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, pages 456–472, New York, NY, USA, 1975. ACM.

[43] D. Tiwari, S. S. Vazhkudai, Y. Kim, X. Ma, S. Boboila, and P. J. Desnoyers. Reducing data movement costs using energy efficient, active computation on SSD. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, HotPower '12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[44] TPC-B. http://www.tpc.org/tpcb/.

[45] P. Willmann, H. Kim, S. Rixner, and V. Pai. An efficient programmable 10 gigabit ethernet network interface card. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 96–107, Feb 2005.

[46] XDD version 6.5. http://www.ioperformance.com/.

[47] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

Physical Disentanglement in a Container-Based File System

Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Department of Computer Sciences
University of Wisconsin, Madison
{ll, yupu, thanhdo, samera, dusseau, remzi}@cs.wisc.edu

Abstract

We introduce IceFS, a novel file system that separates physical structures of the file system. A new abstraction, the *cube*, is provided to enable the grouping of files and directories inside a physically isolated container. We show three major benefits of cubes within IceFS: localized reaction to faults, fast recovery, and concurrent file-system updates. We demonstrate these benefits within a VMware-based virtualized environment and within the Hadoop distributed file system. Results show that our prototype can significantly improve availability and performance, sometimes by an order of magnitude.

1 Introduction

Isolation is central to increased reliability and improved performance of modern computer systems. For example, isolation via virtual address space ensures that one process cannot easily change the memory state of another, thus causing it to crash or produce incorrect results [10].

As a result, researchers and practitioners alike have developed a host of techniques to provide isolation in various computer subsystems: Verghese et al. show how to isolate performance of CPU, memory, and disk bandwidth in SGI's IRIX operating system [58]; Gupta et al. show how to isolate the CPU across different virtual machines [26]; Wachs et al. invent techniques to share storage cache and I/O bandwidth [60]. These are but three examples; others have designed isolation schemes for device drivers [15, 54, 61], CPU and memory resources [2, 7, 13, 41], and security [25, 30, 31].

One aspect of current system design has remained devoid of isolation: the physical on-disk structures of file systems. As a simple example, consider a bitmap, used in historical systems such as FFS [37] as well as many modern file systems [19, 35, 56] to track whether inodes or data blocks are in use or free. When blocks from different files are allocated from the same bitmap, aspects of their reliability are now *entangled*, i.e., a failure in that bitmap block can affect otherwise unrelated files. Similar entanglements exist at all levels of current file systems; for example, Linux Ext3 includes all current update activity into a single global transaction [44], leading to painful and well-documented performance problems [4, 5, 8].

The surprising entanglement found in these systems arises from a central truth: logically-independent file system entities are not physically independent. The result is poor reliability, poor performance, or both.

In this paper, we first demonstrate the root problems caused by physical entanglement in current file systems. For example, we show how a single disk-block failure can lead to global reliability problems, including system-wide crashes and file system unavailability. We also measure how a lack of physical disentanglement slows file system recovery times, which scale poorly with the size of a disk volume. Finally, we analyze the performance of unrelated activities and show they are linked via crash-consistency mechanisms such as journaling.

Our remedy to this problem is realized in a new file system we call *IceFS*. IceFS provides users with a new basic abstraction in which to co-locate logically similar information; we call these containers *cubes*. IceFS then works to ensure that files and directories within cubes are physically distinct from files and directories in other cubes; thus data and I/O within each cube is *disentangled* from data and I/O outside of it.

To realize disentanglement, IceFS is built upon three core principles. First, there should be no shared physical resources across cubes. Structures used within one cube should be distinct from structures used within another. Second, there should be no access dependencies. IceFS separates key file system data structures to ensure that the data of a cube remains accessible regardless of the status of other cubes; one key to doing so is a novel *directory indirection* technique that ensures cube availability in the file system hierarchy despite loss or corruption of parent directories. Third, there should be no bundled transactions. IceFS includes novel *transaction splitting* machinery to enable concurrent updates to file system state, thus disentangling write traffic in different cubes.

One of the primary benefits of cube disentanglement is *localization*: negative behaviors that normally affect all file system clients can be localized within a cube. We demonstrate three key benefits that arise directly from

such localization. First, we show how cubes enable localized *micro-failures*; panics, crashes, and read-only remounts that normally affect the entire system are now constrained to the faulted cube. Second, we show how cubes permit localized *micro-recovery*; instead of an expensive file-system wide check and repair, the disentanglement found at the core of cubes enables IceFS to fully (and quickly) repair a subset of the file system (and even do so online), thus minimizing downtime and increasing availability. Third, we illustrate how transaction splitting allows the file system to commit transactions from different cubes in parallel, greatly increasing performance (by a factor of 2x–5x) for some workloads.

Interestingly, the localization that is innate to cubes also enables a new benefit: *specialization* [17]. Because cubes are independent, it is natural for the file system to tailor the behavior of each. We realize the benefits of specialization by allowing users to choose different journaling modes per cube; doing so creates a performance/consistency knob that can be set as appropriate for a particular workload, enabling higher performance.

Finally, we further show the utility of IceFS in two important modern storage scenarios. In the first, we use IceFS as a host file system in a virtualized VMware [59] environment, and show how it enables fine-grained fault isolation and fast recovery as compared to the state of the art. In the second, we use IceFS beneath HDFS [49], and demonstrate that IceFS provides failure isolation between clients. Overall, these two case studies demonstrate the effectiveness of IceFS as a building block for modern virtualized and distributed storage systems.

The rest of this paper is organized as follows. We first show in Section 2 that the aforementioned problems exist through experiments. Then we introduce the three principles for building a disentangled file system in Section 3, describe our prototype IceFS and its benefits in Section 4, and evaluate IceFS in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Motivation

Logical entities, such as directories, provided by the file system are an illusion; the underlying physical entanglement in file system data structures and transactional mechanisms does not provide true isolation. We describe three problems that this entanglement causes: global failure, slow recovery, and bundled performance. After discussing how current approaches fail to address them, we describe the negative impact on modern systems.

2.1 Entanglement Problems

2.1.1 Global Failure

Ideally, in a robust system, a fault involving one file or directory should not affect other files or directories, the remainder of the OS, or other users. However, in current file systems, a single fault often leads to a *global failure*.

Global Failures	Ext3	Ext4	Btrfs
Crash	129	341	703
Read-only	64	161	89

Table 1: **Global Failures in File Systems.** *This table shows the average number of crash and read-only failures in Ext3, Ext4, and Btrfs source code across 14 versions of Linux (3.0 to 3.13).*

Fault Type	Ext3	Ext4
Metadata read failure	70 (66)	95 (90)
Metadata write failure	57 (55)	71 (69)
Metadata corruption	25 (11)	62 (28)
Pointer fault	76 (76)	123 (85)
Interface fault	8 (1)	63 (8)
Memory allocation	56 (56)	69 (68)
Synchronization fault	17 (14)	32 (27)
Logic fault	6 (0)	17 (0)
Unexpected states	42 (40)	127 (54)

Table 2: **Failure Causes in File Systems.** *This table shows the number of different failure causes for Ext3 and Ext4 in Linux 3.5, including those caused by entangled data structures (in parentheses). Note that a single failure instance may have multiple causes.*

A common approach for handling faults in current file systems is to either *crash* the entire system (e.g., by calling `BUG_ON`, `panic`, or `assert`) or to mark the whole file system *read-only*. Crashes and read-only behavior are not constrained to only the faulty part of the file system; instead, a global reaction is enforced for the whole system. For example, Btrfs crashes the entire OS when it finds an invariant is violated in its extent tree; Ext3 marks the whole file system as read-only when it detects a corruption in a single inode bitmap. To illustrate the prevalence of these coarse reactions, we analyzed the source code and counted the average number of such global failure instances in Ext3 with JBD, Ext4 with JBD2, and Btrfs from Linux 3.0 to 3.13. As shown in Table 1, each file system has hundreds of invocations to these poor global reactions.

Current file systems trigger global failures to react to a wide range of system faults. Table 2 shows there are many root causes: metadata failures and corruptions, pointer faults, memory allocation faults, and invariant faults. These types of faults exist in real systems [11, 12, 22, 33, 42, 51, 52], and they are used for fault injection experiments in many research projects [20, 45, 46, 53, 54, 61]. Responding to these various faults in a non-global manner is non-trivial; the table shows that a high percentage (89% in Ext3, 65% in Ext4) of these faults are caused by entangled data structures (e.g., bitmaps and transactions).

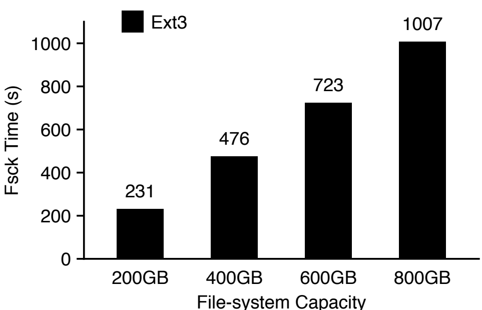


Figure 1: **Scalability of E2fsck on Ext3.** *This figure shows the fsck time on Ext3 with different file-system capacity. We create the initial file-system image on partitions of different capacity (x-axis). We make 20 directories in the root directory and write the same set of files to every directory. As the capacity changes, we keep the file system at 50% utilization by varying the amount of data in the file set.*

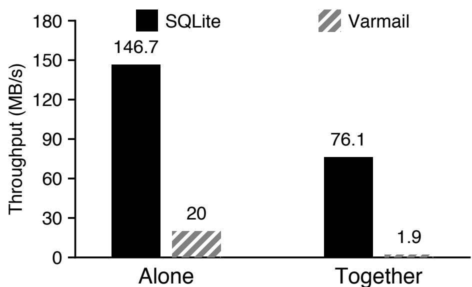


Figure 2: **Bundled Performance on Ext3.** *This figure shows the performance of running SQLite and Varmail on Ext3 in ordered mode. The SQLite workload, configured with write-ahead logging, asynchronously writes 40KB values in sequential key order. The Varmail workload involves 16 threads, each of which performs a series of create-append-sync and read-append-sync operations.*

2.1.2 Slow Recovery

After a failure occurs, file systems often rely on an offline file-system checker to recover [39]. The checker scans the whole file system to verify the consistency of metadata and repair any observed problems. Unfortunately, current file system checkers are *not scalable*: with increasing disk capacities and file system sizes, the time to run the checker is unacceptably long, decreasing availability. For example, Figure 1 shows that the time to run a checker [55] on an Ext3 file system grows linearly with the size of the file system, requiring about 1000 seconds to check an 800GB file system with 50% utilization. Ext4 has better checking performance due to its layout optimization [36], but the checking performance is similar to Ext3 after aging and fragmentation [34].

Despite efforts to make checking faster [14, 34, 43], check time is still constrained by file system size and disk bandwidth. The root problem is that current checkers are *pessimistic*: even though there is only a small piece of corrupt metadata, the entire file system is checked. The main reason is that due to entangled data structures, it is hard or even impossible to determine which part of the file system needs checking.

2.1.3 Bundled Performance and Transactions

The previous two problems occur because file systems fail to isolate metadata structures; additional problems occur because the file system journal is a shared, global data structure. For example, Ext3 uses a generic journaling module, JBD, to manage updates to the file system. To achieve better throughput, instead of creating a separate transaction for every file system update, JBD groups all updates within a short time interval (e.g., 5s) into a single global transaction; this transaction is then committed periodically or when an application calls `fsync()`.

Unfortunately, these bundled transactions cause the performance of independent processes to be bundled. Ideally, calling `fsync()` on a file should flush only the dirty data belonging to that particular file to disk; unfortunately, in the current implementation, calling `fsync()` causes unrelated data to be flushed as well. Therefore, the performance of write workloads may suffer when multiple applications are writing at the same time.

Figure 2 illustrates this problem by running a database application SQLite [9] and an email server workload Varmail [3] on Ext3. SQLite sequentially writes large key/value pairs asynchronously, while Varmail frequently calls `fsync()` after small random writes. As we can see, when these two applications run together, both applications' performance degrades significantly compared with running alone, especially for Varmail. The main reason is that both applications share the same journaling layer and each workload affects the other. The `fsync()` calls issued by Varmail must wait for a large amount of data written by SQLite to be flushed together in the same transaction. Thus, the single shared journal causes performance entanglement for independent applications in the same file system. Note that we use an SSD to back the file system, so device performance is not a bottleneck in this experiment.

2.2 Limitations of Current Solutions

One popular approach for providing isolation in file systems is through the namespace. A namespace defines a subset of files and directories that are made visible to an application. Namespace isolation is widely used for better security in a shared environment to constrain different applications and users. Examples include virtual machines [16, 24], Linux containers [2, 7], `chroot`, BSD `jail` [31], and Solaris Zones [41].

However, these abstractions fail to address the problems mentioned above. Even though a namespace can restrict application access to a subset of the file system, files from different namespaces still share metadata, sys-

tem states, and even transactional machinery. As a result, a fault in any shared structure can lead to a global failure; a file-system checker still must scan the whole file system; updates from different namespaces are bundled together in a single transaction.

Another widely-used method for providing isolation is through static disk partitions. Users can create multiple file systems on separate partitions. Partitions are effective at isolating corrupted data or metadata such that read-only failure can be limited to one partition, but a single `panic()` or `BUG_ON()` within one file system may crash the whole OS, affecting all partitions. In addition, partitions are not flexible in many ways and the number of partitions is usually limited. Furthermore, storage space may not be effectively utilized and disk performance may decrease due to the lack of a global block allocation. Finally, it can be challenging to use and manage a large number of partitions across different file systems and applications.

2.3 Usage Scenarios

Entanglement in the local file system can cause significant problems to higher-level services like virtual machines and distributed file systems. We now demonstrate these problems via two important cases: a virtualized storage environment and a distributed file system.

2.3.1 Virtual Machines

Fault isolation within the local file system is of paramount importance to server virtualization environments. In production deployments, to increase machine utilization, reduce costs, centralize management, and make migration efficient [23, 48, 57], tens of virtual machines (VMs) are often consolidated on a single host machine. The virtual disk image for each VM is usually stored as a single or a few files within the host file system. If a single fault triggered by one of the virtual disks causes the host file system to become read-only (e.g., metadata corruption) or to crash (e.g., assertion failures), then all the VMs suffer. Furthermore, recovering the file system using fsck and redeploying all VMs require considerable downtime.

Figure 3 shows how VMware Workstation 9 [59] running with an Ext3 host file system reacts to a read-only failure caused by one virtual disk image. When a read-only fault is triggered in Ext3, all three VMs receive an error from the host file system and are immediately shut down. There are 10 VMs in the shared file system; each VM has a preallocated 20GB virtual disk image. Although only one VM image has a fault, the entire host file system is scanned by e2fsck, which takes more than eight minutes. This experiment demonstrates that a single fault can affect multiple unrelated VMs; isolation across different VMs is not preserved.

2.3.2 Distributed File Systems

Physical entanglement within the local file system also negatively impacts distributed file systems, especially in multi-tenant settings. Global failures in local file systems manifest themselves as machine failures, which are handled by crash recovery mechanisms. Although data is not lost, fault isolation is still hard to achieve due to long timeouts for crash detection and the layered architecture. We demonstrate this challenge in HDFS [49], a popular distributed file system used by many applications.

Although HDFS provides fault-tolerant machinery such as replication and failover, it does not provide fault isolation for applications. Thus, applications (e.g., HBase [1, 27]) can only rely on HDFS to prevent data loss and must provide fault isolation themselves. For instance, in HBase multi-tenant deployments, HBase servers can manage tables owned by various clients. To isolate different clients, each HBase server serves a certain number of tables [6]. However, this approach does not provide complete isolation: although HBase servers are grouped based on tables, their tables are stored in HDFS nodes, which are not aware of the data they store. Thus, an HDFS server failure will affect multiple HBase servers and clients. Although indirection (e.g., HBase on HDFS) simplifies system management, it makes isolation in distributed systems challenging.

Figure 4 illustrates such a situation: four clients concurrently read different files stored in HDFS when a machine crashes; the crashed machine stores data blocks for all four clients. In this experiment, only the first client is fortunate enough to not reference this crashed node and thus finishes early. The other three lose throughput for 60 seconds before failing over to other nodes. Although data loss does not occur as data is replicated on multiple nodes in HDFS, this behavior may not be acceptable for latency-sensitive applications.

3 File System Disentanglement

To avoid the problems described in the previous section, file systems need to be redesigned to avoid artificial coupling between logical entities and physical realization. In this section, we discuss a key abstraction that enables such disentanglement: the file system cube. We then discuss the key principles underlying a file system that realizes disentanglement: no shared physical resources, no access dependencies, and no bundled transactions.

3.1 The Cube Abstraction

We propose a new file system abstraction, the *cube*, that enables applications to specify which files and directories are logically related. The file system can safely combine the performance and reliability properties of groups of files and their metadata that belong to the same cube; each cube is physically isolated from others and is thus

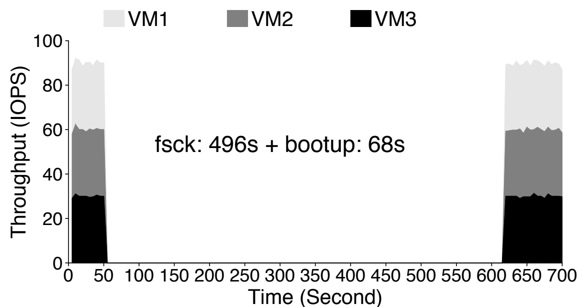


Figure 3: **Global Failure for Virtual Machines.** *This figure shows how a fault in Ext3 affects all three virtual machines (VMs). Each VM runs a workload that writes 4KB blocks randomly to a 1GB file and calls* `fsync()` *after every 10 writes. We inject a fault at 50s, run e2fsck after the failure, and reboot all three VMs.*

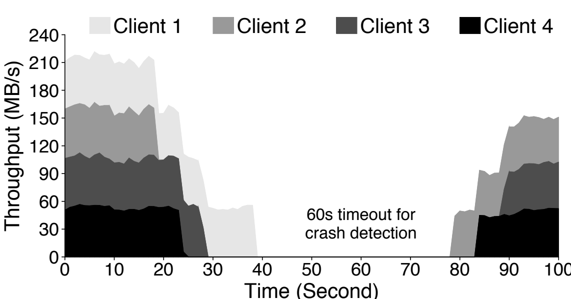


Figure 4: **Impact of Machine Crashes in HDFS.** *This figure shows the negative impact of physical entanglement within local file systems on HDFS. A kernel panic caused by a local file system leads to a machine failure, which negatively affects the throughput of multiple clients.*

completely independent at the file system level.

The cube abstraction is easy to use, with the following operations:

Create a cube: A cube can be created on demand. A default global cube is created when a new file system is created with the `mkfs` utility.

Set cube attributes: Applications can specify customized attributes for each cube. Supported attributes include: failure policy (e.g., read-only or crash), recovery policy (e.g., online or offline checking) and journaling mode (e.g., high or low consistency requirement).

Add files to a cube: Users can create or move files or directories into a cube. By default, files and directories inherit the cube of their parent directory.

Delete files from a cube: Files and directories can be removed from the cube via `unlink`, `rmdir`, and `rename`.

Remove a cube: An application can delete a cube completely along with all files within it. The released disk space can then be used by other cubes.

The cube abstraction has a number of attractive properties. First, each cube is *isolated* from other cubes both logically and physically; at the file system level, each cube is independent for failure, recovery, and journaling. Second, the use of cubes can be *transparent* to applications; once a cube is created, applications can interact with the file system without modification. Third, cubes are *flexible*; cubes can be created and destroyed on demand, similar to working with directories. Fourth, cubes are *elastic* in storage space usage; unlike partitions, no storage over-provision or reservation is needed for a cube. Fifth, cubes can be *customized* for diverse requirements; for example, an important cube may be set with high consistency and immediate recovery attributes. Finally, cubes are *lightweight*; a cube does not require extensive memory or disk resources.

3.2 Disentangled Data Structures

To support the cube abstraction, key data structures within modern file systems must be disentangled. We discuss three principles of disentangled data structures: no shared physical resources, no access dependencies, and no shared transactions.

3.2.1 No Shared Physical Resources

For cubes to have independent performance and reliability, multiple cubes must not share the same physical resources within the file system (e.g., blocks on disk or pages in memory). Unfortunately, current file systems freely co-locate metadata from multiple files and directories into the same unit of physical storage.

In classic Ext-style file systems, storage space is divided into fixed-size *block groups*, in which each block group has its own metadata (i.e., a group descriptor, an inode bitmap, a block bitmap, and inode tables). Files and directories are allocated to particular block groups using heuristics to improve locality and to balance space. Thus, even though the disk is partitioned into multiple block groups, any block group and its corresponding metadata blocks can be shared across any set of files. For example, in Ext3, Ext4 and Btrfs, a single block is likely to contain inodes for multiple unrelated files and directories; if I/O fails for one inode block, then all the files with inodes in that block will not be accessible. As another example, to save space, Ext3 and Ext4 store many group descriptors in one disk block, even though these group descriptors describe unrelated block groups.

This false sharing percolates from on-disk blocks up to in-memory data structures at runtime. Shared resources directly lead to global failures, since a single corruption or I/O failure affects multiple logically-independent files. Therefore, to isolate cubes, a disentangled file system must partition its various data structures into smaller independent ones.

3.2.2 No Access Dependency

To support independent cubes, a disentangled file system must also ensure that one cube does not contain references to or need to access other cubes. Current file systems often contain a number of data structures that violate this principle. Specifically, *linked lists* and *trees* encode dependencies across entries by design. For example, Ext3 and Ext4 maintain an orphan inode list in the super block to record files to be deleted; Btrfs and XFS use Btrees extensively for high performance. Unfortunately, one failed entry in a list or tree affects all entries following or below it.

The most egregious example of access dependencies in file systems is commonly found in the implementation of the *hierarchical directory structure*. In Ext-based systems, the path for reaching a particular file in the directory structure is implicitly encoded in the physical layout of those files and directories on disk. Thus, to read a file, all directories up to the root must be accessible. If a single directory along this path is corrupted or unavailable, a file will be inaccessible.

3.2.3 No Bundled Transactions

The final data structure and mechanism that must be disentangled to provide isolation to cubes are transactions. To guarantee the consistency of metadata and data, existing file systems typically use journaling (e.g., Ext3 and Ext4) or copy-on-write (e.g., Btrfs and ZFS) with transactions. A transaction contains temporal updates from many files within a short period of time (e.g., 5s in Ext3 and Ext4). A shared transaction batches multiple updates and is flushed to disk as a single atomic unit in which either all or none of the updates are successful.

Unfortunately, transaction batching artificially tangles together logically independent operations in several ways. First, if the shared transaction fails, updates to all of the files in this transaction will fail as well. Second, in physical journaling file systems (e.g., Ext3), a `fsync()` call on one file will force data from other files in the same transaction to be flushed as well; this falsely couples performance across independent files and workloads.

4 The Ice File System

We now present IceFS, a file system that provides cubes as its basic new abstraction. We begin by discussing the important internal mechanisms of IceFS, including novel directory independence and transaction splitting mechanisms. Disentangling data structures and mechanisms enables the file system to provide behaviors that are localized and specialized to each container. We describe three major benefits of a disentangled file system (localized reactions to failures, localized recovery, and specialized journaling performance) and how such benefits are realized in IceFS.

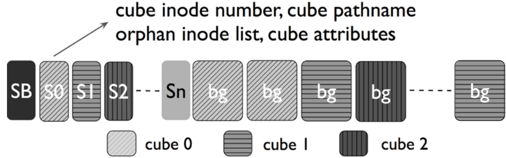


Figure 5: **Disk Layout of IceFS.** *This figure shows the disk layout of IceFS. Each cube has a sub-super block, stored after the global super block. Each cube also has its own separated block groups. Si: sub-super block for cube i; bg: a block group.*

4.1 IceFS

We implement a prototype of a disentangled file system, IceFS, as a set of modifications to Ext3, a standard and mature journaling file system in many Linux distributions. We disentangle Ext3 as a proof of concept; we believe our general design can be applied to other file systems as well.

4.1.1 Realizing the Cube Abstraction

The cube abstraction does not require radical changes to the existing POSIX interface. In IceFS, a cube is implemented as a special directory; all files and sub-directories within the cube directory belong to the same cube.

To create a cube, users pass a cube flag when they call `mkdir()`. IceFS creates the directory and records that this directory is a cube. When creating a cube, customized cube attributes are also supported, such as a specific journaling mode for different cubes. To delete a cube, only `rmdir()` is needed.

IceFS provides a simple mechanism for filesystem isolation so that users have the freedom to define their own policies. For example, an NFS server can automatically create a cube for the home directory of each user, while a VM server can isolate each virtual machine in its own cube. An application can use a cube as a data container, which isolates its own data from other applications.

4.1.2 Physical Resource Isolation

A straightforward approach for supporting cubes is to leverage the existing concept of a block group in many existing file systems. To disentangle shared resources and isolate different cubes, IceFS dictates that a block group can be assigned to only one cube at any time, as shown in Figure 5; in this way, all metadata associated with a block group (e.g., bitmaps and inode tables) belongs to only one cube. A block group freed by one cube can be allocated to any other cube. Compared with partitions, the allocation unit of cubes is only one block group, much smaller than the size of a typical multiple GB partition.

When allocating a new data block or an inode for a cube, the target block group is chosen to be either an empty block group or a block group already belonging to the cube. Enforcing the requirement that a block group

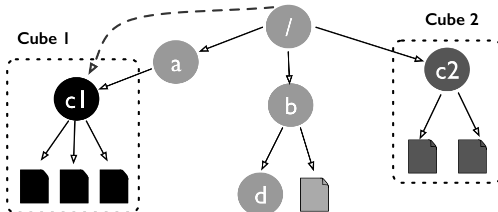


Figure 6: **An Example of Cubes and Directory Indirection.** *This figure shows how the cubes are organized in a directory tree, and how the directory indirection for a cube is achieved.*

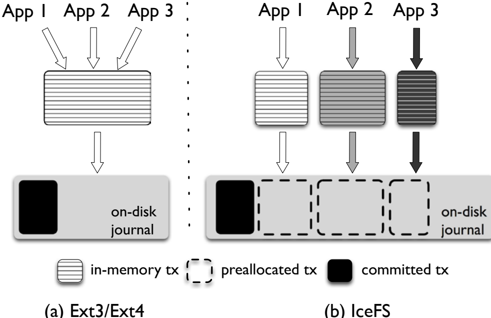


Figure 7: **Transaction Split Architecture.** *This figure shows the different transaction architectures in Ext3/4 and IceFS. In IceFS, different colors represent different cubes' transactions.*

is devoted to a single cube requires changing the file and directory allocation algorithms such that they are cube-aware without losing locality.

To identify the cube of a block group, IceFS stores a cube ID in the group descriptor. To get the cube ID for a file, IceFS simply leverages the static mapping of inode numbers to block groups as in the base Ext3 file system; after mapping the inode of the file to the block group, IceFS obtains the cube ID from the corresponding group descriptor. Since all group descriptors are loaded into memory during the mount process, no extra I/O is required to determine the cube of a file.

IceFS trades disk and memory space for the independence of cubes. To save memory and reduce disk I/O, Ext3 typically places multiple contiguous group descriptors into a single disk block. IceFS modifies this policy so that only group descriptors from the same cube can be placed in the same block. This approach is similar to the meta-group of Ext4 for combining several block groups into a larger block group [35].

4.1.3 Access Independence

To disentangle cubes, no cube can reference another cube. Thus, IceFS partitions each global list that Ext3 maintains into per-cube lists. Specifically, Ext3 stores the head of the global orphan inode list in the super block. To isolate this shared list and the shared super block, IceFS uses one *sub-super* block for each cube; these sub-super blocks are stored on disk after the super block and each references its own orphan inode list as shown in Figure 5. IceFS preallocates a fixed number of sub-super blocks following the super block. The maximum number of sub-super blocks is configurable at `mkfs` time. These sub-super blocks can be replicated within the disk similar to the super block to avoid catastrophic damage of sub-super blocks.

In contrast to a traditional file system, if IceFS detects a reference from one cube to a block in another cube, then it knows that reference is incorrect. For example, no data block should be located in a different cube than the inode of the file to which it belongs.

To disentangle the file namespace from its physical representation on disk and to remove the naming dependencies across cubes, IceFS uses *directory indirection*, as shown in Figure 6. With directory indirection, each cube records its top directory; when the file system performs a pathname lookup, it first finds a longest prefix match of the pathname among the cubes' top directory paths; if it does, then only the remaining pathname within the cube is traversed in the traditional manner. For example, if the user wishes to access `/home/bob/research/paper.tex` and `/home/bob/research/` designates the top of a cube, then IceFS will skip directly to parsing `paper.tex` within the cube. As a result, any failure outside of this cube, or to the `home` or `bob` directories, will not affect accessing `paper.tex`.

In IceFS, the path lookup process performed by the VFS layer is modified to provide directory indirection for cubes. The inode number and the pathname of the top directory of a cube are stored in its sub-super block; when the file system is mounted, IceFS pins in memory this information along with the cube's dentry, inode, and pathname. Later, when a pathname lookup is performed, VFS passes the pathname to IceFS so that IceFS can check whether the pathname is within any cube. If there is no match, then VFS performs the lookup as usual; otherwise, VFS uses the matched cube's dentry as a shortcut to resolve the remaining part of the pathname.

4.1.4 Transaction Splitting

To disentangle transactions belonging to different cubes, we introduce *transaction splitting*, as shown in Figure 7. With transaction splitting, each cube has its own running transaction to buffer writes. Transactions from different cubes are committed to disk in parallel without any waiting or dependencies across cubes. With this approach, any failure along the transaction I/O path can be attributed to the source cube, and the related recovery action can be triggered only for the faulty cube, while other healthy cubes still function normally.

IceFS leverages the existing generic journaling mod-

ule of Ext3, JBD. To provide specialized journaling for different cubes, each cube has a virtual journal managed by JBD with a potentially customized journaling mode. When IceFS starts an atomic operation for a file or directory, it passes the related cube ID to JBD. Since each cube has a separate virtual journal, a commit of a running transaction will only be triggered by its own `fsync()` or timeout without any entanglement with other cubes.

Different virtual journals share the physical journal space on disk. At the beginning of a commit, IceFS will first reserve journal space for the transaction of the cube; a separate committing thread will flush the transaction to the journal. Since transactions from different cubes write to different places on the journal, IceFS can perform multiple commits in parallel. Note that, the original JBD uses a shared lock to synchronize various structures in the journaling layer, while IceFS needs only a single shared lock to allocate transaction space; the rest of the transaction operations can now be performed independently without limiting concurrency.

4.2 Localized Reactions to Failures

As shown in Section 2, current file systems handle serious errors by crashing the whole system or marking the entire file system as read-only. Once a disentangled file system is partitioned into multiple independent cubes, the failure of one cube can be detected and controlled with a more precise boundary. Therefore, failure isolation can be achieved by transforming a global failure to a local per-cube failure.

4.2.1 Fault Detection

Our goal is to provide a new fault-handling primitive, which can localize global failure behaviors to an isolated cube. This primitive is largely orthogonal to the issue of detecting the original faults. We currently leverage existing detection mechanism within file systems to identify various faults.

For example, file systems tend to detect metadata corruption at the I/O boundary by using their own semantics to verify the correctness of file system structures; file systems check error conditions when interacting with other subsystems (e.g., failed disk read/writes or memory allocations); file systems also check assertions and invariants that might fail due to concurrency problems.

IceFS modifies the existing detection techniques to make them cube-aware. For example, Ext3 calls `ext3_error()` to mark the file system as read-only on an inode bitmap read I/O fault. IceFS instruments the fault-handling and crash-triggering functions (e.g., `BUG_ON()`) to include the ID of the responsible cube; pinpointing the faulty cube is straightforward as all metadata is isolated. Thus, IceFS has cube-aware fault detectors.

One can argue that the incentive for detecting problems in current file systems is relatively low because many of the existing recovery techniques (e.g., calling `panic()`) are highly pessimistic and intrusive, making the entire system unusable. A disentangled file system can contain faults within a single cube and thus provides incentive to add more checks to file systems.

4.2.2 Localized Read-Only

As a recovery technique, IceFS enables a single cube to be made read-only. In IceFS, only files within a faulty cube are made read-only, and other cubes remain available for both reads and writes, improving the overall availability of the file system. IceFS performs this per-cube reaction by adapting the existing mechanisms within Ext3 for making all files read-only.

To guarantee read-only for all files in Ext3, two steps are needed. First, the transaction engine is immediately shut down. Existing running transactions are aborted, and attempting to create a new transaction or join an existing transaction results in an error code. Second, the generic VFS super block is marked as read-only; as a result, future writes are rejected.

To localize read-only failures, a disentangled file system can execute two similar steps. First, with the transaction split framework, IceFS individually aborts the transaction for a single cube; thus, no more transactions are allowed for the faulty cube. Second, the faulty cube alone is marked as read-only, instead of the whole file system. When any operation is performed, IceFS now checks this per-cube state whenever it would usually check the super block read-only state. As a result, any write to a read-only cube receives an error code, as desired.

4.2.3 Localized Crashes

Similarly, IceFS is able to localize a crash for a failed cube, such that the crash does not impact the entire operating system or operations of other cubes. Again, IceFS leverages the existing mechanisms in the Linux kernel for dealing with crashes caused by `panic()`, `BUG()`, and `BUG_ON()`. IceFS performs the following steps:

- *Fail the crash-triggering thread:* When a thread fires an assertion failure, IceFS identifies the cube being accessed and marks that cube as crashed. The failed thread is directed to the failure path, during which the failed thread will free its allocated resources (e.g., locks and memory). IceFS adds this error path if it does not exist in the original code.
- *Prevent new threads:* A crashed cube should reject any new file-system request. IceFS identifies whether a request is related to a crashed cube as early as possible and return appropriate error codes to terminate the related system call. Preventing new accesses consists of blocking the entry point functions and the directory indirection functions. For example, the state of a cube is checked at all the callbacks provided by Ext3, such as super block

operations (e.g., `ext3_write_inode()`), directory operations (e.g., `ext3_readdir()`), and file operations (e.g., `ext3_sync_file()`). One complication is that many system calls use either a pathname or a file descriptor as an input; VFS usually translates the pathname or file descriptor into an inode. However, directory indirection in IceFS can be used to quickly prevent a new thread from entering the crashed cube. When VFS conducts the directory indirection, IceFS will see that the pathname belongs to a crashed cube and VFS will return an appropriate error code to the application.

- *Evacuate running threads:* Besides the crash-triggering thread, other threads may be accessing the same cube when the crash happens. IceFS waits for these threads to leave the crashed cube, so they will free their kernel and file-system resources. Since the cube is marked as crashed, these running threads cannot read or write to the cube and will exit with error codes. To track the presence of on-going threads within a cube, IceFS maintains a simple counter for each cube; the counter is incremented when a system call is entered and decremented when a system call returns, similar to the system-call gate [38].
- *Clean up the cube:* Once all the running threads are evacuated, IceFS cleans up the memory states of the crashed cube similar to the unmount process. Specifically, dirty file pages and metadata buffers belonging to the crashed are dropped without being flushed to disk; clean states, such as cached dentries and inodes, are freed.

4.3 Localized Recovery

As shown in Section 2, current file system checkers do not scale well to large file systems. With the cube abstraction, IceFS can solve this problem by enabling per-cube checking. Since each cube represents an independent fault domain with its own isolated metadata and no references to other cubes, a cube can be viewed as a basic checking unit instead of the whole file system.

4.3.1 Offline Checking

In a traditional file-system checker, the file system must be offline to avoid conflicts with a running workload. For simplicity, we first describe a per-cube offline checker.

Ext3 uses the utility e2fsck to check the file system in five phases [39]. IceFS changes e2fsck to make it cube-aware; we call the resulting checker ice-fsck. The main idea is that IceFS supports partial checking of a file system by examining only faulty cubes. In IceFS, when a corruption is detected at run time, the error identifying the faulty cube is recorded in fixed locations on disk. Thus, when ice-fsck is run, erroneous cubes can be easily identified, checked, and repaired, while ignoring the rest of the file system. Of course, ice-fsck can still perform a full file system check and repair, if desired.

Specifically, ice-fsck identifies faulty cubes and their corresponding block groups by reading the error codes recorded in the journal. Before loading the metadata from a block group, each of the five phases of ice-fsck first ensures that this block group belongs to a faulty cube. Because the metadata of a cube is guaranteed to be self-contained, metadata from other cubes not need to be checked. For example, because an inode in one cube cannot point to an indirect block stored in another cube (or block group), ice-fsck can focus on a subset of the block groups. Similarly, checking the directory hierarchy in ice-fsck is simplified; while e2fsck must verify that every file can be connected back to the root directory, ice-fsck only needs to verify that each file in a cube can be reached from the entry points of the cube.

4.3.2 Online Checking

Offline checking of a file system implies that the data will be unavailable to important workloads, which is not acceptable for many applications. A disentangled file system enables on-line checking of faulty cubes while other healthy cubes remain available to foreground traffic, which can greatly improve the availability of the whole service.

Online checking is challenging in existing file systems because metadata is shared loosely by multiple files; if a piece of metadata must be repaired, then all the related files should be frozen or repaired together. Coordinating concurrent updates between the checker and the file system is non-trivial. However, in a disentangled file system, the fine-grained isolation of cubes makes online checking feasible and efficient.

We note that online checking and repair is a powerful recovery mechanism compared to simply crashing or marking a cube read-only. Now, when a fault or corruption is identified at runtime with existing detection techniques, IceFS can unmount the cube so it is no longer visible, and then launch ice-fsck on the corrupted cube while the rest of the file system functions normally. In our implementation, the on-line ice-fsck is a user-space program that is woken up by IceFS informed of the ID of the faulty cubes.

4.4 Specialized Journaling

As described previously, disentangling journal transactions for different cubes enables write operations in different cubes to proceed without impacting others. Disentangling journal transactions (in conjunction with disentangling all other metadata) also enables different cubes to have different consistency guarantees.

Journaling protects files in case of system crashes, providing certain consistency guarantees, such as metadata

or data consistency. Modern journaling file systems support different modes; for example, Ext3 and Ext4 support, from lowest to highest consistency: *writeback*, *ordered*, and *data*. However, the journaling mode is enforced for the entire file system, even though users and applications may desire differentiated consistency guarantees for their data. Transaction splitting enables a specialized journaling protocol to be provided for each cube.

A disentangled file system is free to choose customized consistency modes for each cube, since there are no dependencies across them; even if the metadata of one cube is updated inconsistently and a crash occurs, other cubes will not be affected. IceFS supports five consistency modes, from lowest to highest: *no fsync*, *no journal*, *writeback journal*, *ordered journal* and *data journal*. In general, there is an incentive to choose modes with lower consistency to achieve higher performance, and an incentive to choose modes with higher consistency to protect data in the presence of system crashes.

For example, a cube that stores important configuration files for the system may use data journaling to ensure both data and metadata consistency. Another cube with temporary files may be configured to use *no journal* (i.e., behave similarly to Ext2) to achieve the highest performance, given that applications can recreate the files if a crash occurs. Going one step further, if users do not care about the durability of data of a particular application, the *no fsync* mode can be used to ignore `fsync()` calls from applications. Thus, IceFS gives more control to both applications and users, allowing them to adopt a customized consistency mode for their data.

IceFS uses the existing implementations within JBD to achieve the three journaling modes of writeback, ordered, and data. Specifically, when there is an update for a cube, IceFS uses the specified journaling mode to handle the update. For *no journal*, IceFS behaves like a non-journaled file system, such as Ext2, and does not use the JBD layer at all. Finally, for *no fsync*, IceFS ignores `fsync()` system calls from applications and directly returns without flushing any related data or metadata.

4.5 Implementation Complexity

We added and modified around 6500 LOC to Ext3/JBD in Linux 3.5 for the data structures and journaling isolation, 970 LOC to VFS for directory indirection and crash localization, and 740 LOC to e2fsprogs 1.42.8 for file system creation and checking. The most challenging part of the implementation was to isolate various data structures and transactions for cubes. Once we carefully isolated each cube (both on disk and in memory), the localized reactions to failures and recovery was straightforward to achieve.

Workload	Ext3 (MB/s)	IceFS (MB/s)	Difference
Sequential write	98.9	98.8	0%
Sequential read	107.5	107.8	+0.3%
Random write	2.1	2.1	0%
Random read	0.7	0.7	0%
Fileserver	73.9	69.8	-5.5%
Varmail	2.2	2.3	+4.5%
Webserver	151.0	150.4	-0.4%

Table 3: **Micro and Macro Benchmarks on Ext3 and IceFS.** *This table compares the throughput of several micro and macro benchmarks on Ext3 and IceFS. Sequential write/read are writing/reading a 1GB file in 4KB requests. Random write/read are writing/reading 128MB of a 1GB file in 4KB requests. Fileserver has 50 threads performing creates, deletes, appends, whole-file writes, and whole-file reads. Varmail emulates a multi-threaded mail server. Webserver is a multi-threaded read-intensive workload.*

5 Evaluation of IceFS

We present evaluation results for IceFS. We first evaluate the basic performance of IceFS through a series of micro and macro benchmarks. Then, we show that IceFS is able to localize many failures that were previously global. All the experiments are performed on machines with an Intel(R) Core(TM) i5-2500K CPU (3.30 GHz), 16GB memory, and a 1TB Hitachi Deskstar 7K1000.B hard drive, unless otherwise specified.

5.1 Overall Performance

We assess the performance of IceFS with micro and macro benchmarks. First, we mount both file systems in the default ordered journaling mode, and run several micro benchmarks (sequential read/write and random read/write) and three macro workloads from Filebench (Fileserver, Varmail, and Webserver). For IceFS, each workload uses one cube to store its data. Table 3 shows the throughput of all the benchmarks on Ext3 and IceFS. From the table, one can see that IceFS performs similarly to Ext3, indicating that our disentanglement techniques incur little overhead.

IceFS maintains extra structures for each cube on disk and in memory. For each cube IceFS creates, one sub-super block (4KB) is allocated on disk. Similar to the original super block, sub-super blocks are also cached in memory. In addition, each cube has its own journaling structures (278 B) and cached running states (104 B) in memory. In total, for each cube, its disk overhead is 4 KB and memory overhead is less than 4.5 KB.

5.2 Localize Failures

We show that IceFS converts many global failures into local, per-cube failures. We inject faults into core file-system structures where existing checks are capable of detecting the problem. These faults are selected from

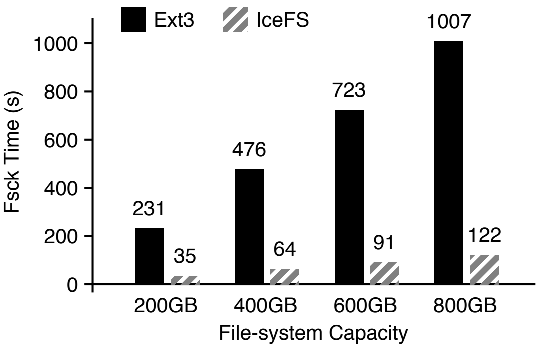


Figure 8: **Performance of IceFS Offline Fsck.** *This figure compares the running time of offline fsck on ext3 and on IceFS with different file-system size.*

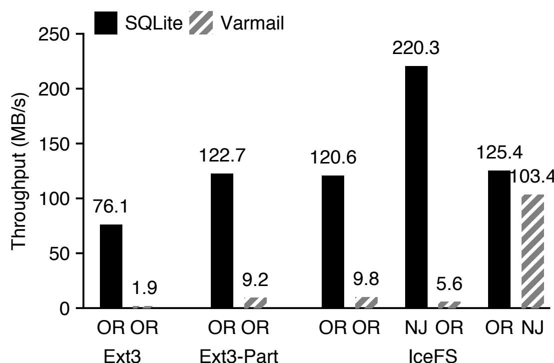


Figure 9: **Running Two Applications on IceFS with Different Journaling Mode.** *This figure compares the performance of simultaneously running SQLite and Varmail on Ext3, partitions and IceFS. In Ext3, both applications run in ordered mode (OR). In Ext3-Part, two separated Ext3 run in ordered mode (OR) on two partitions. In IceFS, two separate cubes with different journaling modes are used: ordered mode (OR) and no-journal mode (NJ).*

Table 2 and they cover all different fault types, including memory allocation failures, metadata corruption, I/O failures, `NULL` pointers, and unexpected states. To compare the behaviors, the faults are injected in the same locations for both Ext3 and IceFS. Overall, we injected nearly 200 faults. With Ext3, in every case, the faults led to global failures of some kind (such as an OS panic or crash). IceFS, in contrast, was able to localize the triggered faults in every case.

However, we found that there are also a small number of failures during the mount process, which are impossible to isolate. For example, if a memory allocation failure happens when initializing the super block during the mount process, then the mount process will exit with an error code. In such cases, both Ext3 and IceFS will not be able to handle it because the fault happens before the file system starts running.

5.3 Fast Recovery

With localized failure detection, IceFS is able to perform offline fsck only on the faulted cube. To measure fsck performance on IceFS, we first create file system images in the same way as described in Figure 1, except that we make 20 cubes instead of directories. We then fail one cube randomly and measure the fsck time. Figure 8 compares the offline fsck time between IceFS and Ext3. The fsck time of IceFS increases as the capacity of the cube grows along with the file system size; in all cases, fsck on IceFS takes much less time than Ext3 because it only needs to check the consistency of one cube.

5.4 Specialized Journaling

We now demonstrate that a disentangled journal enables different consistency modes to be used by different applications on a shared file system. For these experiments, we use a Samsung 840 EVO SSD (500GB) as the underlying storage device. Figure 9 shows the throughput of running two applications, SQLite and Varmail, in Ext3, two separated Ext3 on partitions (Ext3-Part) and IceFS. When running with Ext3 and ordered journaling (two leftmost bars), both applications achieve low performance because they share the same journaling layer and both workloads affect the other. When the applications run with IceFS on two different cubes, their performance increases significantly since `fsync()` calls to one cube do not force out dirty data to the other cube. Compared with Ext3-Part, we can find that IceFS achieves great isolation for cubes at the file system level, similar to running two different file systems on partitions.

We also demonstrate that different applications can benefit from different journaling modes; in particular, if an application can recover from inconsistent data after a crash, the no-journal mode can be used for much higher performance while other applications can continue to safely use ordered mode. As shown in Figure 9, when either SQLite or Varmail is run on a cube with no journaling, that application receives significantly better throughput than it did in ordered mode; at the same time, the competing application using ordered mode continues to perform better than with Ext3. We note that the ordered competing application may perform slightly worse than it did when both applications used ordered mode due to increased contention for resources outside of the file system (i.e., the I/O queue in the block layer for the SSD); this demonstrates that isolation must be provided at all layers of the system for a complete solution. In summary, specialized journaling modes can provide great flexibility for applications to make trade-offs between their performance and consistency requirements.

5.5 Limitations

Although IceFS has many advantages as shown in previous sections, it may perform worse than Ext3 in certain

Device	Ext3 (MB/s)	Ext3-Part (MB/s)	IceFS (MB/s)
SSD	40.8	30.6	35.4
Disk	2.8	2.6	2.7

Table 4: **Limitation of IceFS On Cache Flush.** *This table compares the aggregated throughput of four Varmail instances on Ext3 and IceFS. Each Varmail instance runs in a directory of Ext3, an Ext3 partition (Ext3-Part), or a cube of IceFS. We run the same experiment on both a SSD and hard disk.*

extreme cases. The main limitation of our implementation is that IceFS uses a separate journal commit thread for every cube. The thread issues a device cache `flush` command at the end of every transaction commit to make sure the cached data is persistent on device; this cache flush is usually expensive [21]. Therefore, if many active cubes perform journal commits at the same time, the performance of IceFS may be worse than Ext3 that only uses one journal commit thread for all updates. The same problem exists in separated file systems on partitions.

To show this effect, we choose Varmail as our testing workload. Varmail utilizes multiple threads; each of these threads repeatedly issues small writes and calls `fsync()` after each write. We run multiple instances of Varmail in different directories, partitions or cubes to generate a large number of transaction commits, stressing the file system.

Table 4 shows the performance of running four Varmail instances on our quad-core machine. When running on an SSD, IceFS performs worse than Ext3, but a little better than Ext3 partitions (Ext3-Part). When running on a hard drive, all three setups perform similarly. The reason is that the cache flush time accounts for a large percentage of the total I/O time on an SSD, while the seeking time dominates the total I/O time on a hard disk. Since IceFS and Ext3-Part issue more cache flushes than Ext3, the performance penalty is amplified on the SSD.

Note that this style of workload is an extreme case for both IceFS and partitions. However, compared with separated file systems on partitions, IceFS is still a single file system that can utilize all the related semantic information of cubes for further optimization. For example, IceFS can pass per-cube hints to the block layer, which can optimize the cache flush cost and provide other performance isolation for cubes.

5.6 Usage Scenarios

We demonstrate that IceFS improves overall system behavior in the two motivational scenarios initially introduced in Section 2.3: virtualized environments and distributed file systems.

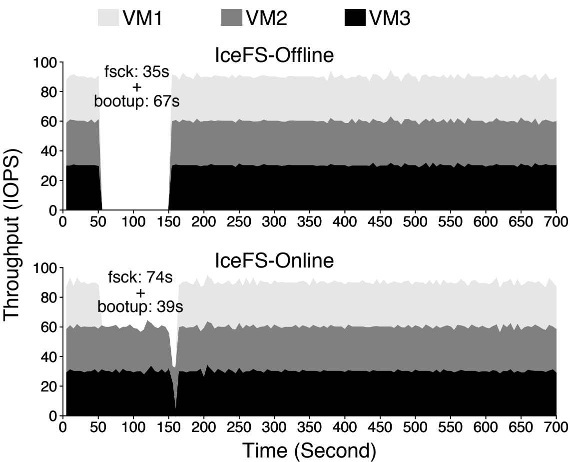


Figure 10: **Failure Handling for Virtual Machines.** *This figure shows how IceFS handles failures in a shared file system which supports multiple virtual machines.*

5.6.1 Virtual Machines

To show that IceFS enables virtualized environments to isolate failures within a particular VM, we configure each VM to use a separate cube in IceFS. Each cube stores a 20GB virtual disk image, and the file system contains 10 such cubes for 10 VMs. Then, we inject a fault to one VM image that causes the host file system to be read-only after 50 seconds.

Figure 10 shows that IceFS greatly improves the availability of the VMs compared to that in Figure 3 using Ext3. The top graph illustrates IceFS with offline recovery. Here, only one cube is read-only and crashes; the other two VMs are shut down properly so the offline cube-aware check can be performed. The offline check of the single faulty cube requires only 35 seconds and booting the three VMs takes about 67 seconds; thus, after only 150 seconds, the three virtual machines are running normally again.

The bottom graph illustrates IceFS with online recovery. In this case, after the fault occurs in VM1 (at roughly 50 seconds) and VM1 crashes, VM2 and VM3 are able to continue. At this point, the online fsck of IceFS starts to recover the disk image file of VM1 in the host file system. Since fsck competes for disk bandwidth with the two running VMs, checking takes longer (about 74 seconds). Booting the single failed VM requires only 39 seconds, but the disk activity that arises as a result of booting competes with the I/O requests of VM2 and VM3, so the throughput of VM2 and VM3 drops for that short time period. In summary, these two experiments demonstrate that IceFS can isolate file system failures in a virtualized environment and significantly reduce system recovery time.

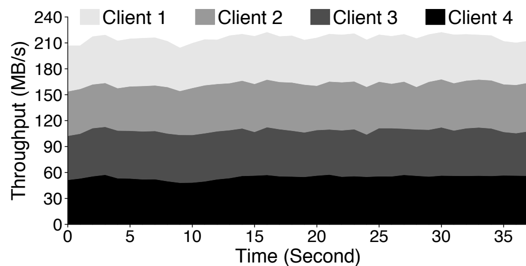


Figure 11: **Impact of Cube Failures in HDFS.** *This figure shows the throughput of 4 different clients when a cube failure happens at time 10 second. Impact of the failure to the clients' throughput is negligible.*

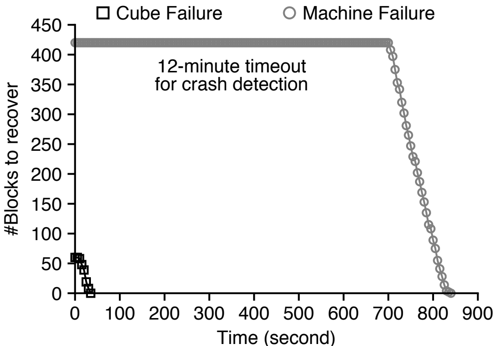


Figure 12: **Data Block Recovery in HDFS.** *The figure shows the number of lost blocks to be regenerated over time in two failure scenarios: cube and whole machine failure. Cube failure results into less blocks to recover in less time.*

5.6.2 Distributed File System

We illustrate the benefits of using IceFS to provide flexible fault isolation in HDFS. Obtaining fault isolation in HDFS is challenging, especially in multi-tenant settings, primarily because HDFS servers are not aware of the data they store, as shown in Section 2.3.2. IceFS provides a natural solution for this problem. We use separate cubes to store different applications' data on HDFS servers. Each cube isolates the data from one application to another; thus, a cube failure will not affect multiple applications. In this manner, IceFS provides end-to-end isolation for applications in HDFS. We added 161 lines to storage node code to make HDFS IceFS-compatible and aware of application data. We do not change any recovery code of HDFS. Instead, IceFS turns global failures (e.g., kernel panic) into partial failures (i.e., cube failure) and leverages HDFS recovery code to handle them. This facilitates and simplifies our implementation.

Figure 11 shows the benefits of IceFS-enabled application-level fault isolation. Here, four clients concurrently access different files stored in HDFS when a cube that stores data for Client 2 fails and becomes inaccessible. Other clients are completely isolated from the cube failure. Furthermore, the failure negligibly impacts the throughput of the client as it does not manifest as machine failure. Instead, it results in a soft error to HDFS, which then immediately isolates the faulty cube and returns an error code the client. The client then quickly fails over to other healthy copies. The overall throughput is stable for the entire workload, as opposed to 60-second period of losing throughput as in the case of whole machine failure described in Section 2.3.2.

In addition to end-to-end isolation, IceFS provides scalable recovery as shown in Figure 12. In particular, IceFS helps reduce network traffic required to regenerate lost blocks, a major bandwidth consumption factor in large clusters [47]. When a cube fails, IceFS again returns an error code to the host server, which then immediately triggers a block scan to find out data blocks that are under-replicated and regenerates them. The number of blocks to recover is proportional to the cube size. Without IceFS, a kernel panic in local file system manifests as whole machine failure, causing a 12-minute timeout for crash detection and making the number of blocks lost and to be regenerated during recovery much larger. In summary, IceFS helps improve not only flexibility in fault isolation but also efficiency in failure recovery.

6 Related Work

IceFS has derived inspiration from a number of projects for improving file system recovery and repair, and for tolerating system crashes.

Many existing systems have improved the reliability of file systems with better recovery techniques. Fast checking of the Solaris UFS [43] has been proposed by only checking the working-set portion of the file system when failure happens. Changing the I/O pattern of the file system checker to reduce random requests has been suggested [14, 34]. A background fsck in BSD [38] checks a file system snapshot to avoid conflicts with the foreground workload. WAFL [29] employs Wafliron [40], an online file system checker, to perform online checking on a volume but the volume being checked cannot be accessed by users. Our recovery idea is based on the cube abstraction which provides isolated failure, recovery and journaling. Under this model, we only check the faulty part of the file system without scanning the whole file system. The above techniques can be utilized in one cube to further speedup the recovery process.

Several repair-driven file systems also exist. Chunkfs [28] does a partial check of Ext2 by partitioning the file system into multiple chunks; however, files and directory can still span multiple chunks, reducing the independence of chunks. Windows ReFS [50] can automatically recover corrupted data from mirrored

storage devices when it detects checksum mismatch. Our earlier work [32] proposes a high-level design to isolate file system structures for fault and recovery isolation. Here, we extend that work by addressing both reliability and performance issues with a real prototype and demonstrations for various applications.

Many ideas for tolerating system crashes have been introduced at different levels. Microrebooting [18] partitions a large application into rebootable and stateless components; to recover a failed component, the data state of each component is persistent in a separate store outside of the application. Nooks [54] isolates failures of device drivers from the rest of the kernel with separated address spaces for each target driver. Membrane [53] handles file system crashes transparently by tracking resource usage and the requests at runtime; after a crash, the file system is restarted by releasing the in-use resources and replaying the failed requests. The Rio file cache [20] protects the memory state of the file system across a system crash, and conducts a warm reboot to recover lost updates. Inspired by these ideas, IceFS localizes a file system crash by microisolating the file system structures and microrebooting a cube with a simple and light-weight design. Address space isolation technique could be used in cubes for better memory fault isolation.

7 Conclusion

Despite isolation of many components in existing systems, the file system still lacks physical isolation. We have designed and implemented IceFS, a file system that achieves physical disentanglement through a new abstraction called cubes. IceFS uses cubes to group logically related files and directories, and ensures that data and metadata in each cube are isolated. There are no shared physical resources, no access dependencies, and no bundled transactions among cubes.

Through experiments, we demonstrate that IceFS is able to localize failures that were previously global, and recover quickly using localized online or offline fsck. IceFS can also provide specialized journaling to meet diverse application requirements for performance and consistency. Furthermore, we conduct two cases studies where IceFS is used to host multiple virtual machines and is deployed as the local file system for HDFS data nodes. IceFS achieves fault isolation and fast recovery in both scenarios, proving its usefulness in modern storage environments.

Acknowledgments

We thank the anonymous reviewers and Nick Feamster (our shepherd) for their tremendous feedback. We thank the members of the ADSL research group for their suggestions and comments on this work at various stages. We thank Yinan Li for the hardware support, and Ao Ma for discussing fsck in detail.

This material was supported by funding from NSF grants CCF-1016924, CNS-1421033, CNS-1319405, and CNS-1218405 as well as generous donations from Amazon, Cisco, EMC, Facebook, Fusion-io, Google, Huawei, IBM, Los Alamos National Laboratory, MdotLabs, Microsoft, NetApp, Samsung, Sony, Symantec, and VMware. Lanyue Lu is supported by the VMWare Graduate Fellowship. Samer Al-Kiswany is supported by the NSERC Postdoctoral Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Apache HBase. https://hbase.apache.org/.

[2] Docker: The Linux Container Engine. https://www.docker.io.

[3] Filebench. http://sourceforge.net/projects/filebench.

[4] Firefox 3 Uses fsync Excessively. https://bugzilla.mozilla.org/show_bug.cgi?id=421482.

[5] Fsyncers and Curveballs. http://shaver.off.net/diary/2008/05/25/fsyncers-and-curveballs/.

[6] HBase User Mailing List. http://hbase.apache.org/mail-lists.html.

[7] Linux Containers. https://linuxcontainers.org/.

[8] Solving the Ext3 Latency Problem. http://lwn.net/Articles/328363/.

[9] SQLite. https://sqlite.org.

[10] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.8 edition, 2014.

[11] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.

[12] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 223–238, San Jose, California, February 2008.

[13] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[14] Eric J. Bina and Perry A. Emrath. A Faster fsck for BSD Unix. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '89)*, San Diego, California, January 1989.

[15] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, Massachusetts, June 2010.

[16] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[17] Calton Pu and Tito Autrey and Andrew Black and Charles Consel and Crispin Cowan and Jon Inouye and Lakshmi Kethana and Jonathan Walpole and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[18] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 31–44, San Francisco, California, December 2004.

[19] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[20] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[21] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[22] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[23] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[24] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[25] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium (Sec '96)*, San Jose, California, 1996.

[26] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006)*, Melbourne, Australia, Nov 2006.

[27] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[28] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep '06)*, Seattle, Washington, November 2006.

[29] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[30] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. Application-Level Isolation and Recovery with Solitude. In *Proceedings of the EuroSys Conference (EuroSys '08)*, Glasgow, Scotland UK, March 2008.

[31] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Second International System Administration and Networking Conference (SANE '00)*, May 2000.

[32] Lanyue Lu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Fault Isolation And Quick Recovery in Isolation File Systems. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, CA, June 2013.

[33] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[34] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[35] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge and, and Laurent Vivier. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[36] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[37] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[38] Marshall Kirk McKusick. Running 'fsck' in the Background. In *Proceedings of BSDCon 2002 (BSDCon '02)*, San Fransisco, California, February 2002.

[39] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[40] NetApp. Overview of WAFL_check. `http://uadmin.nl/init/?p=900`, Sep. 2011.

[41] Oracle Inc. Consolidating Applications with Oracle Solaris Containers. `http://www.oracle.com/technetwork/server-storage/solaris/documentation/consolidating-apps-163572.pdf`, Jul 2011.

[42] Nicolas Palix, Gael Thomas, Suman Saha, Christophe Calves, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*, Newport Beach, California, March 2011.

[43] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast Consistency Checking for the Solaris File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, pages 77–89, New Orleans, Louisiana, June 1998.

[44] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of

Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[45] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[46] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs As Allergies. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[47] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, CA, June 2013.

[48] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[49] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[50] Steven Sinofsky. Building the Next Generation File System for Windows: ReFS. http://blogs.msdn.com/b/b8/archive/2012/01/16/building-the-next-generation-file-system-for-windows-refs.aspx, Jan. 2012.

[51] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, Montreal, Canada, June 1991.

[52] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *Proceedings of the 22st International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 475–484, Boston, USA, July 1992.

[53] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[54] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[55] Theodore Ts'o. http://e2fsprogs.sourceforge.net, June 2001.

[56] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[57] Satyam B. Vaghani. Virtual Machine File System. *ACM SIGOPS Operating Systems Review*, 44(4):57–70, Dec 2010.

[58] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, California, October 1998.

[59] VMware Inc. VMware Workstation. http://www.vmware.com/products/workstation, Apr 2014.

[60] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[61] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

Customizable and Extensible Deployment for Mobile/Cloud Applications

Irene Zhang Adriana Szekeres Dana Van Aken Isaac Ackerman
Steven D. Gribble* Arvind Krishnamurthy Henry M. Levy
University of Washington

Abstract

Modern applications face new challenges in managing today's highly distributed and heterogeneous environment. For example, they must stitch together code that crosses smartphones, tablets, personal devices, and cloud services, connected by variable wide-area networks, such as WiFi and 4G. This paper describes Sapphire*, a distributed programming platform that simplifies the programming of today's mobile/cloud applications. Sapphire's key design feature is its distributed runtime system, which supports a flexible and extensible* deployment layer *for solving complex distributed systems tasks, such as fault-tolerance, code-offloading, and caching. Rather than writing distributed systems code, programmers choose* deployment managers *that extend Sapphire's kernel to meet their applications' deployment requirements. In this way, each application runs on an underlying platform that is customized for its own distribution needs.*

1 Introduction

In less than a decade, the computing landscape has undergone two revolutionary changes: the development of small, yet remarkably powerful, mobile devices and the move to massive-scale cloud computing. These changes have led to a shift away from traditional desktop applications to modern mobile/cloud applications.

As a consequence, modern applications have become inherently *distributed*, with data and code spread across cloud backends and user devices such as phones and tablets. Application programmers face new challenges that were visible only to designers of large-scale distributed systems in the past. Among them are coordinating shared data across multiple devices and servers, offloading code from devices to the cloud, and integrating heterogeneous components with vastly different software stacks and hardware resources.

To address these challenges, programmers must make numerous distributed *deployment* decisions, such as:

- Where data and computation should be located
- What data should be replicated or cached
- What data consistency level is needed

These decisions depend on application requirements – such as scalability and fault tolerance – which force difficult performance vs. function trade-offs. The dependency between application requirements and deployment decisions leads programmers to mix deployment decisions with complex application logic in the code, which makes mobile/cloud applications difficult to implement, debug, maintain, and evolve. Even worse, the rapid evolution of devices, networks, systems, and applications means that the trade-offs that impact these deployment decisions are constantly in flux. For all of these reasons, programmers need *a flexible* system that allows them to easily *create and modify distributed application deployments without needing to rewrite major parts of their application*.

This paper presents Sapphire, a general-purpose distributed programming platform that greatly simplifies the design and implementation of applications spanning mobile devices and clouds. Sapphire removes much of the complexity of managing a wide-area, multi-platform environment, yet still provides developers with the fine-grained control needed to meet critical application needs. A key concept of Sapphire's design is the *separation of application logic from deployment logic*. That is, deployment code is factored out of application code, allowing the programmer to focus on the application logic. At the same time, the programmer has full control over deployment decisions and the flexibility to customize them.

Sapphire's architecture facilitates this separation with a highly extensible distributed kernel/runtime system. At the bottom layer, Sapphire's *Deployment Kernel* (DK) integrates heterogeneous mobile devices and cloud servers through a set of common low-level mechanisms, including best-efforts RPC communication, failure detection, and location finding. Between the kernel and the application is a *deployment layer* – a collection of pluggable *Deployment Manager* (DM) modules that extend the kernel to support application-specific deployment needs, such as replication and caching. DMs are written in a generic, application-transparent way, using interposition to intercept important application events, such as RPC calls. The DK provides a simple yet powerful distributed execution environment and API for DMs that makes them extremely easy to write and extend. Conceptually, Sapphire's DK/DM architecture creates a seamless distributed runtime system that is customized specifically for each application's requirements.

We implemented a Sapphire prototype on Linux servers and Android mobile phones and tablets. The prototype includes a library of 26 Deployment Managers

*Currently at Google.

supporting a wide range of distributed management tasks, such as consistent client-side caching, durable transactions, Paxos replication, and dynamic code offloading between mobile devices and the cloud. We also built 10 Sapphire applications, including a fully featured Twitter clone, a multi-player game, and a shared text editor.

Our experience and evaluation show that Sapphire's extensible three-layer architecture greatly simplifies the construction of both mobile/cloud applications and distributed deployment functions. For example, a single-line application code change – switching from one DM to another – is sufficient to transform a cloud-based multi-player game into a P2P (device-to-device) version that significantly improves the game's performance. The division of function between the DK and DM layers makes deployments extremely easy to code; e.g., the DM to support Paxos state machine replication is only 129 lines of code, an order of magnitude smaller than a C++ implementation built atop an RPC library. We also demonstrate that Sapphire's structure provides fine-grained control over performance trade-offs, delivering performance commensurate with today's popular communication mechanisms like REST.

The next section provides background on current mobile/cloud applications and discusses related work. Section 3 overviews Sapphire and its core distributed runtime system. Section 4 presents the application programming model. Section 5 details the design of the Deployment Kernel, while Section 6 focuses on Deployment Managers, which extend the DK with custom distributed deployment mechanisms. Sapphire's prototype implementation is described in Section 7 and evaluated in Section 8, and we conclude in Section 10.

2 Motivation and Background

Figure 1 shows the deployment of a typical mobile/cloud application. Currently, programmers must deploy applications across a patchwork of user devices, cloud servers, and backend services, while satisfying demanding requirements such as responsiveness and availability. For example, programmers may need to apply caching techniques, perform application-specific code splitting across clients and servers, and develop solutions for fast and convenient data sharing, scalability, and fault tolerance.

Programmers use tools and systems when they match the needs of their application. In some cases an existing system might support an application entirely; for example, a simple application that only requires data synchronization could use a backend storage service like Dropbox [23], Parse [53] or S3 [58]. More complex applications, though, must integrate multiple tools and systems into a custom platform that meets their needs. These systems include server-side storage like Redis [56] or MySQL [49] for fault-tolerance, protocols such as REST [25] and SOAP [62] or libraries like Java RMI and Thrift [3] for distributed communication, load-balanced servers for scalability, client-side caching for lower wide-area latency, and systems for notification [1], coordination [9, 33], and monitoring [18].

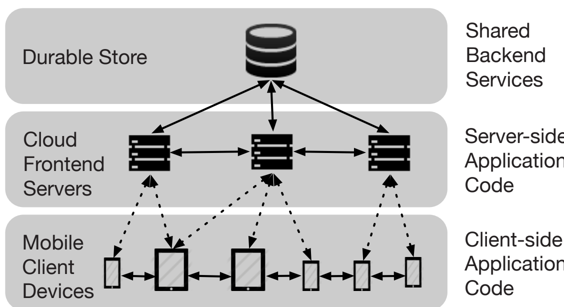


Figure 1: Code for today's applications spans cloud servers and mobile devices. Client-side code runs on varied mobile platforms, while server-side code runs in the cloud, typically using shared backend services like distributed storage.

Sapphire provides a flexible environment whose extension mechanism can subsume the functions of many of these systems, or can integrate them into the platform in a transparent way. Programmers can easily customize the runtime system to meet the needs of their applications. In addition, programmers can quickly switch deployment solutions to respond to environment or requirement changes, or simply to test and compare alternatives during development. Finally, Sapphire's Deployment Manager framework simplifies the development or extension of distributed deployment code.

3 Sapphire Overview

Sapphire is a distributed programming platform designed for flexibility and extensibility. In this section, we cover our goals in designing Sapphire, the deployment model that we assume, and Sapphire's system architecture.

3.1 Design Goals

We designed Sapphire with three primary goals:

1. *Create a distributed programming platform spanning devices and the cloud.* A common platform integrates the heterogeneous distributed environment and simplifies communications, code/data mobility, and replication.
2. *Separate application logic from deployment logic.* The application code is focused on servicing client requests rather than distribution. This simplifies programming, evolution, and optimization.
3. *Facilitate system extension and customization.* The delegation of distribution management to an extensible deployment layer gives programmers the flexibility to easily make or change deployment options.

Sapphire is designed to deploy applications across mobile devices and cloud servers. This environment causes

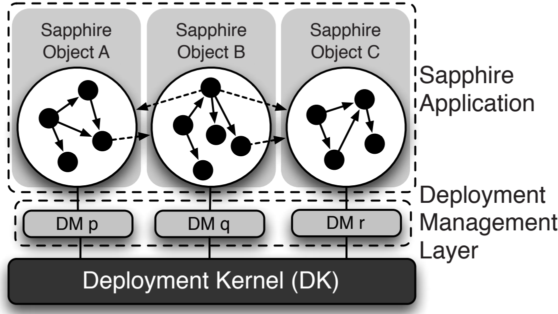


Figure 2: Sapphire runtime architecture. A Sapphire application consists of a distributed collection of Sapphire Objects executing on a distributed Deployment Kernel (DK). A DK instance runs on every device or cloud node. The Deployment Management (DM) layer handles distribution management/deployment tasks, such as replication, scalability, and performance.

significant complexity, as the programmer must stitch together a distributed collection of highly heterogeneous software and hardware components with a broad spectrum of capabilities, while still meeting application goals.

Sapphire is *not* designed for deploying backend services like Spanner [16] or ZooKeeper [33]; its applications interact with such backend services using direct calls, similar to current apps. A Sapphire Deployment Manager can easily integrate a backend service transparently to the application, e.g., using ZooKeeper for coordination or Spanner for fault-tolerance. Sapphire is also not designed for building user interfaces; we expect applications to customize their user interfaces for the devices they employ.

3.2 System Architecture

Figure 2 shows an application-level view of Sapphire's architecture. A Sapphire application, which encompasses all of the client-side and server-side application logic, consists of a collection of *Sapphire Objects* (SOs). Each Sapphire Object functions as a single unit of distribution, like a virtual node. Sapphire Objects in an application share a logical address space that spans all cloud servers and client-side devices. That is, a Sapphire application is written so that all SOs can invoke each other directly through simple location-independent procedure calls.

The bottom layer of Figure 2 is the *Deployment Kernel* (DK), which is a flexible and extensible distributed runtime system. It provides only the most basic distribution functions, including SO addressing and location tracking, best effort RPC-based communication, SO migration, and basic resource management. It does *not* support more complex tasks, such as fault tolerance, failure management, reliability, and consistency. In this way, the DK resembles IP-level network messaging – it is a basic service that relies on higher levels of software to meet more demanding program goals. The kernel is thus deployment agnostic and does not favor (or limit the application to) any specific approaches to deployment issues.

More complex management tasks are supported in the deployment layer by extensions to the DK, called *Deployment Managers* (DMs). Each Sapphire Object can optionally have an attached DM – shown in the middle of Figure 2 – which provides runtime distribution support in addition to the minimal features of the DK. The programmer selects a DM to manage each SO; e.g., he may choose a DM that handles failures to improve fault-tolerance, or one to cache data locally on a mobile device for performance. We have built a library of DMs supporting common distribution tasks used by applications today.

The separation between the DK and DMs provides significant flexibility and extensibility within the Sapphire distributed programming platform. As extensions to the DK, Deployment Managers provide additional distribution management features or guarantees for individual SOs. Often, these features involve performance trade-offs; thus, not every application or every SO will want or need a DM. Finally, by separating application logic (in the application program) from deployment logic (provided by DMs), we greatly reduce application complexity and allow programmers to easily change application deployment or performance behaviors.

4 Programming Model

The Sapphire application programming model is object based and could be integrated with any object-oriented language. Our implementation (Section 7) uses Java.

Sapphire Objects are the key programming abstraction for managing application code and data *locality*. To develop a Sapphire application, the programmer first builds the application logic as a single object-oriented program. He then breaks the application into distributed components by declaring a set of application objects to be Sapphire Objects. Sapphire Objects can still call each other via normal method invocation, however, these calls may now be remote invocations. Finally, the programmer applies Deployment Managers (DMs) to SOs as desired for additional distributed management features. In this section, we will show that the Sapphire programming model provides: (1) ease of programming in a distributed environment, (2) flexibility in deployment, and (3) programmer control over performance.

Defining Sapphire Objects. Programmers define Sapphire Objects as classes using a `sapphireclass` declaration, instead of the standard `class` declaration. As an example, Figure 3 shows a code snippet from our Twitter-clone, BlueBird. All instances of the User class defined here are independent SOs. In this case, the programmer has also specified a DM for the class, called ConsistentCaching, to enhance the object's performance.

SOs can encapsulate internal language-defined objects

```
1   public sapphireclass User uses ConsistentCaching {
2
3       // user handle
4       String username;
5       // people who follow me
6       User[] followers;
7       // people who I follow
8       User[] friends;
9         ....
10      public String getUsername() {
11          return username;
12      }
13      public User[] getMyFollowers() {
14          return followers;
15      }
16      public User[] getPeopleIFollow() {
17          return friends;
18      }
19      public Tweet[] getMyTweets() {
20          return myTweets.getTweets();
21      }
22  }
```

Figure 3: Example Sapphire object from BlueBird.

(Java objects in our system), such as the User string and arrays. These are shown as small solid circles in Figure 2; the solid arrows in the figure are references between internal objects within an SO. SO-internal objects cannot move independently or be accessed directly from outside the SO. The SO is therefore the granularity of distribution and decomposition in Sapphire. Moving an SO always moves all of its internal objects along with it; therefore, the programmer knows that all SO-internal objects will always be co-located with the SO.

A Sapphire Object encapsulates data and computation into a "virtual node" that: (1) ensures that each data/computation unit (a Sapphire Object) will always have its code and data on the same node, (2) lets the system transparently relocate or offload that unit, (3) supports easy replication of units, and (4) provides an easy-to-understand unit of failure and recovery. These benefits make Sapphire Objects a powerful abstraction; using fine-grained programmer-defined Sapphire Objects, instead of a coarse-grained client/server architecture, increases *both* flexibility in distributed deployment and programmer control over performance.

Calling Sapphire Objects. Sapphire Objects communicate using method invocation. The dashed lines in Figure 2 show cross-SO references, which are used to invoke the target SO's public methods. Invocation is location-independent and *symmetric*; it can occur transparently from mobile device to server, from server to device, from device to device, or between servers in the cloud. An SO can be moved by its DM or by the DK as a result of resource constraints on the executing node. Therefore, between two consecutive invocations from SO A to SO B, either or both objects can change location; the DK hides this change from the communicating parties. Invocations can fail, e.g., due to network or node failure; DMs help to handle failure on behalf of SOs.

SOs are passed by reference. All other arguments and return values from SO invocations are passed by value. For example, the return value of `getUsername()` in Figure 3 is a copy of the `username` object stored inside the SO, while `getMyFollowers()` returns a copy of the array containing references to User SOs. This preserves the encapsulation and isolation properties of Sapphire Objects, since it is impossible to export the address of internal objects within them.

Our goal was to create a uniform programming model integrating mobile devices and the cloud *without* hiding performance costs and trade-offs from the programmer. Therefore, the programmer makes explicit choices in the decomposition of the application into SOs; once that choice is made, the system provides location-independent communication, which simplifies programming in the distributed environment.

Choosing Deployment Managers. Programmers employ the `uses` keyword to specify a DM when defining a Sapphire Object. For example, in Figure 3, the `sapphireclass` declaration (line 1) binds the `ConsistentCaching` DM to the `User` class. In this case, every instance of `User` created by the program will have the `ConsistentCaching` DM attached to it. It is easy to change the DM binding with a simple change to the sapphireclass definition.

Supporting DMs on a class basis lets programmers specify different features or properties for different application components. While the binding between an SO and its DM could be specified outside of the language (e.g., through a configuration file), we felt that this choice should be visible in the code because deployment decisions about the SO are closely tied to the requirements of an SO.

Sapphire provides a library of standard DMs, and most programmers will be able to choose the behavior they want from the standard library. Additionally, DMs are extensible; we discuss the API for building them in the next section. As programmers can build their own DMs and DMs are designed to be reusable, we expect the library to grow naturally over time.

An SO can have at most one DM, and each instance of the SO must use the same DM. We chose these restrictions for simplicity and predictability, both in the design of applications and DMs. In particular, the behavior of multiple DMs attached to an SO depends on the order in which the functions of the multiple DMs are invoked, and DMs could potentially interfere with each other. For this reason, programmers achieve the same result by explicitly composing DMs using inheritance. This allows the programmer to precisely control the actions of the composed DM. Since instances of the same SO should have the same deployment requirements, we chose not to allow different DMs for different instances of the same SO.

DMs separate management code into generic, reusable

modules that: (1) automatically deploy the application in complex ways, (2) give programmers per-application-component control over deployment trade-offs, and (3) allow programmers to easily change deployment decisions. These advantages make DMs a powerful mechanism for deploying distributed applications.

5 Deployment Kernel

Sapphire's *Deployment Kernel* is a distributed runtime system for Sapphire applications. At a high level, the goal of the DK is to create an integrated execution platform across mobile devices and servers. The key functions provided by the DK include: (1) management and location tracking of Sapphire Objects, (2) location-transparent inter-object communications (RPC), (3) low-level replica support, and (4) services to simplify the writing and execution of Deployment Managers.

A DK instance provides best-effort deployment of a single Sapphire application. It consists of a set of servers that run on every mobile and back-end computing device used by the application, and a centralized Object Tracking System (OTS) for tracking Sapphire Objects.

The Sapphire OTS is a distributed, fault-tolerant coordination service, similar to Chubby [9], ZooKeeper [33] and Tango [4]. The OTS is responsible for tracking Sapphire Objects across DK servers. DK servers only communicate occasionally with the OTS when creating or moving SOs. DK servers do not have to contact the OTS on every RPC because SO references contain a cached copy of the SO's last location,

Each DK server hosts a number of SOs by acting as an event server for the SOs, receiving and dispatching RPCs. The DK server also hosts and manages the DMs for those SOs. DK servers instantiate SOs locally by initializing the SO's memory, creating its DM (which potentially has components on multiple nodes), and registering the SO with the OTS. Once created, the server can move the SO at any time because SO location and movement are invisible to the application.

The DK provides primitive SO scheduling and placement. If a DK server becomes overloaded, it will contact the OTS to find a new server to host the SO, move the SO to the new server, and update the OTS with the SO's new location. The DK API, described in Section 6, provides primitives that allow DMs to express more complex placement and scheduling policies, such as geo-replicated fault-tolerance, load balancing, etc.

To route an RPC to an SO, the calling DK server sends the RPC request to the destination server cached in the SO reference. If the destination no longer hosts the SO, the caller contacts the OTS to obtain the new address. If the destination server is unavailable, the calling server returns an error, because RPC in the DK is always best effort; DMs implement more advanced RPC handling, like retrying RPCs, routing RPCs between replicas, etc.

DK servers are not fault-tolerant: when they fail, they simply reboot. That is, on recovery, DK servers do not recover the SOs that they hosted on failure; they simply register with the OTS and begin hosting new SOs. Failures are entirely handled by DMs. We assume there is a failure detection system, such as FALCON [39], to notify the OTS when servers fail, which will then notify the DMs of the SOs that were hosted on the failed server.

We expect devices to be Internet connected most of the time, since applications today frequently depend on online access to cloud servers. When a device becomes disconnected, its DK server continues to run, but the application will be unable to make or receive remote RPCs. Any SOs hosted on a disconnected device will thus be inaccessible to outside devices and servers. The OTS keeps a list of mobile device IP addresses to quickly re-register SOs hosted on those devices when they reconnect. DMs can provide more advanced offline access.

6 Deployment Managers

A key feature of the Sapphire kernel is its support for the programming and execution of Deployment Managers, which customize and control the behavior of individual SOs in the distributed mobile/cloud environment. The DK provides direct API support for DMs. That API is available to DM developers, who we expect to be more technically sophisticated than application developers, although the DM framework can be used by anyone to customize or build new DMs. As this section will show, DMs can accomplish complex distributed deployment tasks with surprisingly little code. This is due to the careful factoring of function between the DMs and the DK: the DK does the heavy lifting, while the DMs simply tell the DK what to lift through the DK's API.

6.1 DM Library

Sapphire provides programmers with a library of DMs that encompass many management features, including controls over placement and RPC semantics, fault-tolerance, load balancing and scaling, code-offloading, and peer-to-peer deployment. Table 1 lists the DMs that we have built along with a description and the LoC count (from SLOCCount [71]) for each one. We built these DMs both to provide programmers with useful DMs for their applications and to illustrate the flexibility and programming ease of the DM programming framework.

6.2 DM Structure and API

We designed the DM API to provide as minimal an interface as possible while still supporting a wide range of extensions. A DM extends the functionality of the DK to meet the deployment requirements of a specific SO by interposing on DK events for the SO. For example, on an RPC to the SO, the DK will make an upcall into the DM

Table 1: Library of Deployment Managers

Category	Extension	Description	LoC
Primitives	Immutable	Efficient distribution and access for immutable SOs	19
	AtLeastOnceRPC	Automatically retry RPCs for bounded amount of time	27
	KeepInPlace	Keep SO where it was created (e.g., to access device-specific APIs)	15
	KeepInCloud	Keep SO on cloud server (e.g., for availability)	15
	KeepOnDevice	Keep SO on accessing client device and dynamically move	45
Caching	ExplicitCaching	Caching w/ explicit push and pull calls from the application	41
	LeaseCaching	Caching w/ server granting leases, local reads and writes for lease-holder	133
	WriteThroughCaching	Caching w/ writes serialized on the server and stale, local reads	43
	ConsistentCaching	Caching w/ updates sent to every replica for strict consistency	98
Serializability	SerializableRPC	Serialize all RPCs to SO with server-side locking	10
	LockingTransactions	Multi-RPC transactions w/ locking, no concurrent transactions	81
	OptimisticTransactions	Transactions with optimistic concurrency control, abort on conflict	92
Checkpointing	ExplicitCheckpoint	App-controlled checkpointing to disk, revert last checkpoint on failure	51
	PeriodicCheckpoint	Checkpoint to disk every N RPCs, revert to last checkpoint on failure	65
	DurableSerializableRPC	Durable serializable RPCs, revert to last successful RPC on failure	29
	DurableTransactions	Durably committed transactions, revert to last commit on failure	112
Replication	ConsensusRSM-Cluster	Single cluster replicated SO w/ atomic RPCs across at least $f+1$ replicas	129
	ConsensusRSM-Geo	Geo-replicated SO w/ atomic RPCs across at least $f+1$ replicas	132
	ConsensusRSM-P2P	SO replicated across client devices w/ atomic RPCs over $f+1$ replicas	138
Mobility	ExplicitMigration	Dynamic placement of SO with explicit move call from application	20
	DynamicMigration	Adaptive, dynamic placement to minimize latency based on accesses	57
	ExplicitCodeOffloading	Dynamic code offloading with offload call from application	49
	CodeOffloading	Adaptive, dynamic code offloading based on measured latencies	95
Scalability	LoadBalancedFrontEnd	Simple load balancing w/ static number of replicas and no consistency	53
	ScaleUpFrontEnd	Load-balancing w/ dynamic allocation of replicas and no consistency	88
	LoadBalancedMasterSlave	Dynamic allocation of load-balanced M-S replicas w/ eventual consistency	177

for that SO. DMs are implemented as objects, therefore each DM can execute code on each upcall and store state between upcalls.

A DM consists of three component types: the *Proxy*, the *Instance Manager*, and the *Coordinator*. A programmer builds a DM by defining three object classes, one for each type. Since DMs are intended to manage distribution, the DK creates a distributed execution environment in which they operate; i.e., a DM is *itself* distributed and its components can operate on different nodes. When the DK instantiates a Sapphire Object with an attached DM, it also instantiates and distributes the DM's components. The DK provides transparent RPC between the DM components of an SO instance for coordination between components.

Figure 4 shows an example deployment of the DM components for a *single* Sapphire Object A. The DK may instantiate many Proxies and Instance Managers but at most one Coordinator, as shown in this figure. The center box (marked "Instance A") indicates that A has two replicas, marked replica 1 and replica 2. Each replica has its own copy of the Instance Manager. Were the DM to request a third replica of A, the DK would also create a new Instance Manager for that replica. A replica and its Instance Manager are always located on the same node.

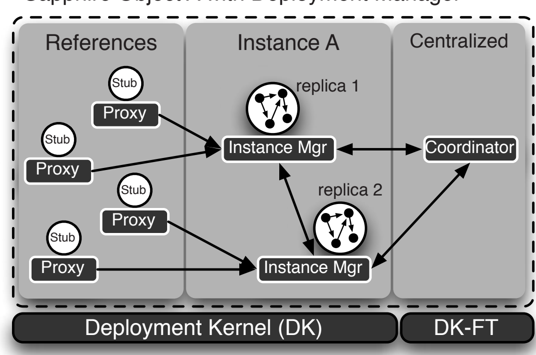


Figure 4: Deployment Manager (DM) organization. The components named *Proxy*, *Instance Mgr*, and *Coordinator* are all part of the DM for one Sapphire Object instance (shown here with two replicas). DK-FT is a set of fault-tolerant DK nodes, which also host the OTS, that support reliable centralized tasks for DMs and the DK.

Each component of the DM is responsible for a particular set of distributed tasks. Proxies are responsible for *caller-side tasks*, like routing method calls. Instance Managers are responsible for *callee-side tasks*, like keeping replicas of the SO synchronized. Note that, due

Table 2: Deployment Managers Upcall API.

Event	Description
`onCreate`	Creation of SO instance
`onRPC`	Method invocation on SO
`onFailure`	Replica failed
`onDestroy`	Coordinator eliminated SO
`onHighLatency`	Avg. RPC latency > limit
`onLowMemory`	Node running out of memory
`onMemberChange`	New replica added to group
`onRefRequest`	Request for an SO reference

Table 3: DK API for Deployment Managers

Operation	Description
`invoke(RPC)`	Invoke RPC on the local SO
`invoke(SO,RPC)`	Invoke RPC on a specific SO
`getNode()`	Get ID for local node
`getNodes()`	Get list of all nodes
`pin(node)`	Move SO to a node.
`setHighLatency(ms)`	Set limit for RPC latency
`durable_put(SO)`	Save copy of the SO
`durable_get(key)`	Retrieve SO
`replicate()`	Create a replica
`destroyReplica(IM)`	Eliminate a replica
`getReplicas()`	Get list of replicas for SO
`getReplica()`	Get ref to SO instance
`setReplica(SO)`	Set ref to SO instance
`copy(SO)`	Create a copy of the SO instance
`diff(SO,SO)`	Diff two SO instances
`sync(SO)`	Synchronize two SO instances
`getIM()`	Get ref to DM Instance Mgr
`setIM(IM)`	Set reference to DM Instance Mgr
`getCoordinator()`	Get ref to DM Coordinator
`getReference(IM)`	Create DM Proxy for IM
`registerMethod(m)`	Register a custom method for DM
`getRegion()`	Get ID for local region
`getNode()`	Get ID for local node
`pin(region)`	Move SO to region
`pin(node)`	Move SO to node
`getRegions()`	Get list of server regions
`getNodes()`	Get list of nodes in local region

to the symmetric nature of SOs, the caller of the method may be on a cloud server and the SO itself may be on a client device. Lastly, the Coordinator is responsible for *centralized tasks* such as failure handling. All three components are optional; a DM can define one or more of the components, and the DK will instantiate only those components that are defined.

The DK completely manages DM components; they run only when invoked, they reside only where the DK places them and are limited to communicating with other components in the same DM instance, which are attached to a single SO. The DK invokes DM components using *upcalls*, which are shown in Table 2. Each component receives a different set of upcalls according to the component's responsibilities. By *interposing* on Sapphire Object events such as method invocations, DMs can implement a variety of distributed management features transparently and generically.

In each upcall, the DM component can perform various management tasks on the SO using a set of primitives supported by the DK. Table 3 lists these primitives. The DM components of an SO instance can communicate directly with each other through a transparent RPC mechanism provided by the DK. Note that the DK supports only the most basic replication functions, namely, creating a new replica for an SO and reporting on replica locations. All decisions about the number of replicas, when to create or delete them, how to synchronize them, and how to handle failures occur at the DM level.

The left-most box in Figure 4 shows four other SOs. Each contains a reference to A, shown as an RPC stub in the figure, to which the DK has attached an instance of A's DM Proxy component. Making an RPC to A through the DK and its DM proceeds as follows. The DK reflects the call via an `onRPC()` upcall to the attached Proxy. The upcall to the Proxy lets A's DM intercept an RPC *on the caller's node* where, for example, it can implement client-local caching. If the Proxy wants to forward the call to replica 1 of A, it simply invokes replica 1's Instance Manager which runs in the same DK server as replica 1. The Instance Manager will pass the RPC through to replica 1 of A.

Because the Proxies and Instance Managers for A are all part of the same Deployment Manager, they all understand whether or not the SO (A, in this case) is replicated, and, if so, *how* that replication is implemented. The choice of which replica to call is made *inside* the DM components, which are aware of each other and can communicate with each other directly through RPCs.

Finally, the DK instantiates one Coordinator for each DM instance, shown in the right-most box of Figure 4. The OTS manages Coordinators, keeping them fault-tolerant and centrally accessible. It is well known that a centralized coordinator can simplify many distributed algorithms (e.g., eliminating the need for leader election). Since the DK needs the OTS to tracking Sapphire Objects, it was easy to provide fault-tolerance for some DMs as well. We do not expect every DM to have a Coordinator, and even if there is a Coordinator, it is used sparingly for management tasks that are easiest handled centrally, such as instantiating new replicas in the event of failures. In this sense, Coordinators are similar to other centralized management systems, like Chubby [9] or ZooKeeper [33].

Programmers can easily extend or compose existing DMs using inheritance. The new DM inherits all of the behavior of the super-DM's Component object classes. The programmer can then override or combine upcalls in each component. While we considered automatic composition,

```
1  public class LeasedCaching extends DManager {
2    public class LCProxy extends Proxy {
3      Lease lease;
4      SapphireObject so;
5
6      public Object onRPC(SapphireRPC rpc) {
7        if (!lease.isValid() || lease.isExpired()) {
8          lease = Sapphire.getReplica().getLease();
9          if (!lease.isValid()) {
10           throw new SONotAvailableException(
11             ``Could not get lease.'');
12         } else {
13           so = lease.getSO();
14         }
15       }
16
17       SapphireObject oldSO = Sapphire.copy(so);
18       Sapphire.invoke(so, rpc);
19       SOStream diff = Sapphire.diff(oldSO, so);
20       if (diff) Sapphire.getReplica().update(diff);
21     }
22   }
23
24   public class LCReplica extends InstanceManager {
25     public synchronized Lease getLease();
26     public synchronized void update(SOStream);
27     // Code for Instance Manager methods
28   }
29 }
```

Figure 5: Example Deployment Manager with arguments.

we believe that the DM programmer should be involved to ensure that the composed DM implements exactly the behavior that the programmer expects. Our experience with composing DMs has shown that the use of inheritance for DM composition is straightforward and intuitive.

6.3 DM Code Example

Figure 5 shows a simplified definition of the LeasedCaching DM that we provide in the Sapphire Library. We include code for the Proxy component and the function declarations from the Instance Manager. This DM does not have a Coordinator because it does not need centralized management.

The LeasedCaching DM is not replicated, so DK will only create one Instance Manager. The Instance Manager hands out mutually exclusive leases to Proxies (which reside with the remote reference to the SO) and uses timeouts to deal with failed Proxies. The Proxy with a valid lease can read or write to a local copy. Read-only operations do not incur communications, which saves latency over a slow network, but updates are synchronously propogated to the Instance Manager in case of Proxy failure.

When the application invokes a method on an SO with this DM attached, the caller's Proxy: (1) verifies that it holds a lease, (2) performs the method call on its local copy, (3) checks whether the object has been modified (using `diff()`), and (4) synchronizes the remote object with its cached copy if the object changed, using an `update()` call to the Instance Manager.

Each Proxy stores the lease in the Lease object (line 3) and a local copy of the Sapphire Object (line 4). If the Proxy does not hold a valid lease, it must get one from the Instance Manager (line 8) before invoking its local SO copy. If the Proxy is not able to get the lease, the DM throws a SONotAvailableException (line 10). The application is prepared for any RPC to an SO to fail, so it will catch the exception and deal with it. The application also knows that the SO uses the LeasedCaching SOM, so it understands the error string (line 11).

If the Proxy is able to get a lease from the Instance Manager, the lease will contain an up-to-date copy of the SO (line 13). The Proxy will make a clean copy of the SO (line 17), invoke the method on its local copy (line 18) and then diff the local copy with the clean copy to check for updates (line 19). If the SO changed, the Proxy will update the Instance Manager's copy of the SO (line 20). The copy and diff is necessary because the Proxy does not know which SO methods might write to the SO, thus requiring an update to the Instance Manager. If the DM had more insight into the SO (i.e., the SO lets the DM know which methods are read-only), we could skip this step.

The example illustrates a few interesting properties of DMs. First, DM code is application agnostic and can perform only a limited set of operations on the SO that it manages. In particular, it can interpose only on method calls to its SO, and it manipulates the managed SO as a black box. For example, there are DMs that automatically cache an SO, but no DMs that cache a part of an SO. This ensures a clean separation of object management code from application logic and allows the DM to be reused across different applications and objects.

Second, a DM cannot span more than one Sapphire Object: it performs operations only on the object that it manages. We chose not to support cross-SO management because it would require the DM to better understand the application; as well, it might cause conflicts between the DMs of different SOs. As a result, there are DMs that provide multi-RPC transactions on a single SO, but we do not support cross-SO transactions. However, the programmer could combine multiple Sapphire Objects into one SO or implement concurrency support at the application level to achieve the same effect.

6.4 DM Design Examples

This section discusses the design and implementation of several classes of DMs from the Sapphire Library, listed in Table 1. Our goal is to show how the DM API can be used to extend the DK for a wide range of distributed management features.

Code-offloading. The code-offloading DMs are useful for compute-intensive applications. The CodeOffloading DM supports transparent object migration based on the performance trade-off between locating an object on a device or in the cloud, while the ExplicitCodeOffloading DM allows the application to decide when to move computation. The ExplicitCodeOffloading DM gives

the application more control than the automated CodeOffloading DM, but is less transparent because the SO must interact with the DM.

Once the DK creates the Sapphire Object on a mobile device, the automated CodeOffloading DM replicates the object in the cloud. The device-side DM Instance Manager then runs several RPCs locally and asks the cloud-side Instance Manager to do the same, calculating the cost of running on each side. An adaptive algorithm, based on Q-learning [70], gradually chooses the lowest-cost option for each RPC. Periodically, the DM retests the alternatives to dynamically adapt to changing behavior since the cost of offloading depends on the type of computation and the network connection, which can change over time.

Peer-to-peer. We built peer-to-peer DMs to support the direct sharing of SOs across client mobile devices *without* needing to go through the cloud. These DMs dynamically place replicas on nodes that contain references to the SO. We implemented the DM using a centralized Coordinator that attempts to place replicas as close to the callers as possible, without exceeding an application-specified maximum number of replicas. We show the performance impact of this P2P scheme in Section 8.

Replication. The Sapphire Library contains three replication DMs that replicate a Sapphire Object across several servers for fault tolerance. They offer guarantees of serializability and exactly-once semantics, along with fault-tolerance. They require that the SO is deterministic and only makes idempotent calls to other SOs.

The Library's replication DMs model the SO as a replicated state machine (RSM) that executes operations on a *master replica*. These DMs all inherit from a common DM that implements the RSM, then extend the common DM to implement different policies for replica placement (e.g., Geo-replicated, P2P).

The RSM DM uses a Coordinator to instantiate the desired number of replicas, designate a leader, and maintain information regarding membership of the replica group. The Coordinator associates an epoch number with this information, which it updates when membership changes.

For each RPC, Instance Managers forward the request to the Instance Manager of the master replica, which logs the RPC and assigns it an ID. The master then sends the ID and epoch number to the other Instance Managers, which accept it if they do not have another RPC with the same ID. If the master receives a response from at least f other Instance Managers, it executes the RPC and synchronizes the state of the SO on the other replicas. If one of the replicas fails, the DK notifies the Coordinator, which allocates a new replica, designates a leader, starts a new epoch, and informs other replicas of the change.

Scalability. To scale Sapphire Objects that handle a large number of requests, the Sapphire Library includes both stateless and stateful scalability DMs. The LoadBalancedFrontEnd DM provides simple load balancing among a set number of replicas. This DM only supports Sapphire Objects that are stateless (i.e., do not require consistency between replicas); however, the SO is free to access state in other Sapphire Objects or on disk. The ScaleUpFrontEnd DM extends the LoadBalancedFrontEnd DM with automatic scale-up, The DM monitors the latency of requests and creates new replicas when the load on the SO and the latency increases. Finally, the LoadBalancedMasterSlave provides scalability for read-heavy workloads by dynamically allocating a number of read-only replicas that receive updates from the master replica. This DM uses the Coordinator to organize replicas and select the master. We show the utility of our scalability DMs in Section 8.

Discussion. The DM's upcall API and its associated DK API are relatively small (only 8 upcalls and 27 DK calls), yet powerful enough to cover a wide range of sophisticated deployment tasks. Most of our DMs are under a hundred lines of code. There are three reasons for this efficiency of expression. First is the division of labor between the DMs and the DK. The DK supports fundamental mechanisms such as RPC, object creation and mobility, and replica management. Therefore, the DK performs the majority of the work in deployment operations, while the DMs simply tell the DK what work to perform.

Second is the availability of a centralized, fault-tolerant Coordinator in the DM environment. This reduces the complexity of many distributed protocols; e.g., in the ConsensusRSM DMs, the Coordinator simplifies consensus by determining the leader and group membership. Our three replication DMs share this code but make different replica placement decisions, meeting different goals and properties with the same mechanism. Inheritance facilitates the composition of new DMs from existing ones; e.g., the DurableTransactions DM builds upon the OptimisticTransactions DM, adding fault-tolerance with only 20 more lines of code.

Finally, the decomposition of applications into Sapphire Objects greatly simplifies DM implementation. We implemented the code-offloading DM in only 95 LoC because we do not have to determine the unit of code to offload dynamically, and because the application provides a hint that the SO is compute-intensive by choosing the DM. In contrast, current code-offloading systems [19, 30, 14] are much more complex because they lack information on application behavior and because the applications are not easily composed into locality units, such as objects.

7 Implementation

Our DK prototype was built using Java to accommodate Android mobile devices. Altogether, the DK consists of 12,735 lines of Java code, including 10,912 lines of

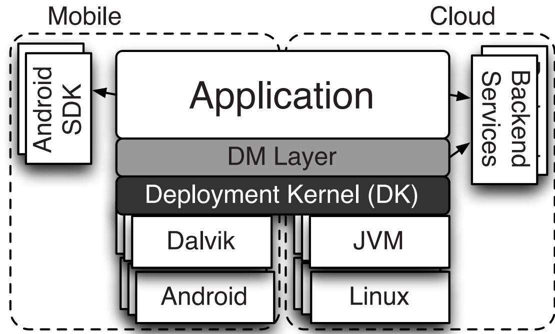


Figure 6: Sapphire application and runtime system.

Apache Harmony RMI code, which we had to port to Dalvik. Dalvik was developed based on Apache Harmony, but does not include an implementation for Java RMI.

Figure 6 shows the prototype's architecture. We used Sun's Java 1.6.0_38 JVM to run Sapphire in the cluster, while the tablets and phones ran Sapphire on the Android 4.2 Dalvik VM. We used Java RMI for low-level RPCs between DK nodes. We used Voldemort [69] as the storage back-end for our checkpointing DMs.

Java RMI provides only point-to-point communication and only supports calls to Java objects that have a special Java RMI-provided interface. Thus, we could only use Java RMI for low-level communication between DK servers and the OTS. To achieve transparent communication between SOs and between DM components, we built a compiler (862 LoC) that creates stubs for SOs and for DM Instance Managers and Coordinators. Having a stub for each SO allows the DK server to route RPCs and invoke DM components on the callee and caller side. DM Instance Managers and Coordinators also require stubs because the DK needs to be able to support transparent RPC from Instance Managers and Proxies. The compiler generates stubs as Java classes that extend the class of the target object, replacing all method contents with forwarding functions into the DK. A stub is therefore a reference that can be used for transparent communication with the remote object through the DK.

We also rely on Apache Harmony's implementation of RMI serialization – with Java reflection to marshall and unmarshall objects – for sending, diffing and copying objects. We did no optimization of Java RMI at all in this prototype. We could have applied well-known techniques [44, 54, 50] to improve RPC performance and expect to do so in the future; however, as we show in our evaluation, our performance is competitive with widely used client-server mechanisms, such as REST. In order to achieve this performance on mobile devices, we had to fix several bugs that caused performance problems in the Apache Harmony RMI code that we ported to Android.

Our prototype does not currently include secure communication between DK servers. Java RMI supports SSL/TLS, so our prototype could easily support encrypted communication between DK servers. We would also require an authentication mechanism for registering DK servers on mobile devices, like Google SSO [28].

In today's applications, mechanisms such as access control checks are typically provided by the application. With a unified programming platform like Sapphire, it becomes possible to move security mechanisms into the platform itself. While this discussion is outside the scope of the paper, we are currently exploring the use of information flow control-based protection for mobile/cloud applications in the context of Sapphire's object and DK/DM structure.

8 Experience and Evaluation

This section presents qualitative and quantitative evaluations of Sapphire. We first describe our experience building new applications and porting applications to Sapphire. Second, we provide low-level DK performance measurements, and an evaluation of several DMs and their performance characteristics. Our experience demonstrates that: (1) Sapphire applications are easy to build, (2) the separation of application code and deployment code, along with the use of symmetric (i.e., non-client-server) communication, maximizes flexibility and choice of deployment for programmers, and (3) Deployment Managers can be used effectively to improve performance and scalability in a dynamic distributed environment.

8.1 Applications

We consider the design and implementation of several Sapphire applications with respect to three objectives:

- **Development Ease:** It should be easy to develop mobile/cloud applications either from scratch or by porting non-distributed mobile device applications to Sapphire. Furthermore, it should be possible to write application code without explicitly addressing distribution management.
- **Deployment Flexibility:** The programmer should be able to choose from alternative distribution management schemes and change deployment decisions without rewriting application code.
- **Management Code Generality:** It should be possible to build generic distribution management components that can be used widely both within an application and across different applications.

Table 4 lists several applications that we built or ported, along with their LoC. We built three applications from scratch: an online to-do list, a collaborative text and table editor, and a multi-player game. We also built a fully-featured Twitter clone, called BlueBird, and paired it with the front-end UI from Twimight [68], an open-source Android Twitter client. The table also lists six non-distributed, compute-intensive applications that

Table 4: Sapphire applications. We divide each application into front-end code (the UI) and back-end code (application logic). The source column indicates whether we developed new native Sapphire code or ported open-source code to Sapphire.

Application	Back-end		Front-end	
	Source	LoC	Source	LoC
To Do List	Native	48	Native	132
Text/Table Editor	Native	409	Native	533
Multi-player Game	Native	588	Native	1,186
BlueBird	Native	783	Ported	13,009
Sudoku Solver	Ported	76	-	-
Regression	Ported	348	-	-
Image Recognition	Ported	102	-	-
Physics Engine	Ported	108	-	-
Calculus	Ported	818	-	-
Chess AI	Ported	427	-	-

we ported to Sapphire.

Development Ease. It took relatively little time and programming experience to develop Sapphire applications. In particular, the existence of a DM library lets programmers write application logic without needing to manage distribution explicitly. Two applications – the multi-player game and collaborative editor – were written by undergraduates who had never built mobile device or web applications and had little distributed systems experience. In under a week, each student wrote a working mobile/cloud application of between 1000 and 1500 lines of code consisting of five or six Sapphire Objects spanning the UI and Sapphire back-end.

Porting existing applications to Sapphire was easy as well. For the compute-intensive applications, a single line change was sufficient to turn a Java object into a distributed SO that could adaptively execute either on the cloud or the mobile device. We did not have to handle failures because the CodeOffload DM hides them by transparently re-executing the computation locally when the remote site is not available. An undergraduate ported all six applications – and implemented the CodeOffload DM as well – in less than a week.

Our largest application was BlueBird, a Twitter clone that was organized as ten Sapphire Objects: Tweet, Tag, TagManager, Timeline, UserTimeline, HomeTimeline, MentionsTimeline, FavoritesTimeline, User and UserManager. We implemented all Twitter functions except for messaging and search in under 800 lines. In comparison, BigBird [22], an open-source Twitter clone, is 2563 lines of code, and Retwis-J [38], which relies heavily on Redis search functionality, is 932 lines of code.

Distributed mobile/cloud applications must cope with the challenges of running on resource-constrained mobile devices, unreliable cloud servers, and high-latency, wide-area links. Using Sapphire, these challenges are handled by selecting DMs from the DM library, which greatly simplifies the programmer's task and makes it easy to develop and test alternative deployments.

Deployment Flexibility. Changing an SO's DM, which changes its distribution properties, requires only a one-line code change. We made use of this property throughout the development of our applications as we experimented with our initial distribution decisions and tried to optimize them.

In BlueBird, for example, we initially chose not to make Tweet and Tag into SOs; since these objects are small and immutable, we thought they did not need to be independent, globally shared objects. Later, we realized that it would be useful to refer directly to Tweets and Tags from Timeline objects rather than accessing them through another SO. We therefore changed them to SOs – a trivial change – and then employed ExplicitCaching for both of them to reduce the network delay for reads of the tweet or tag strings.

As another example, we encountered a deployment decision in the development of our multi-player game. The Game object lasts only for the duration of a game and can be accessed only from two devices used to play. Since the object does not need high reliability or availability, it can be deployed in any number of ways: on a server, on one of the devices, or on both devices. We first deployed the Game object on a cloud server and then decided to experiment with peer-to-peer alternatives. Changing from the cloud deployment to peer-to-peer using the KeepOnDevice and ConsensusRSM-P2P managers in our DM library required only a *single line change*, and improved performance (see Section 8.4) and allowed games to continue when the server is unavailable. In contrast, changing an application for one of today's systems from a cloud deployment to a peer-to-peer mobile device deployment would require significant application rewriting (and might even be impossible without an intermediary cloud component due to the client-server nature of existing systems).

Management Code Generality. We applied several DMs to multiple SOs within individual applications and across applications. For example, many of our applications have an object that is shared among a small number of users or devices (e.g., ToDoList, Document, etc.). To make reads faster while ensuring that users see immediate updates, we used the ConsistentCaching DM for all of these applications. Without the DM structure, the programmers would have to write the caching and synchronization code explicitly for each case.

Even within BlueBird, which has 10 Sapphire Object types, we could reuse several DMs. If the deployment code for each BlueBird SO had to be implemented in the application, the application would grow by at least

800 LoC, more than doubling in size! This number is conservative: it assumes the availability of the DM API and the DK for support. Without those mechanisms, even more code would be required.

8.2 Experimental Setup

Our experiments were performed on a homogeneous cluster of server machines and several types of devices (tablets and phones). Each server contained 2 quad-core Intel Xeon E5335 2.00GHz CPUs with 8GB of DRAM running Ubuntu 12.04 with Linux kernel version 3.2.0-26. The devices were Nexus 7 tablets, which run on a 1.3 GHz quad-core Cortex A9 with 1 GB of DRAM, and Nexus S phones with a 1 GHz single-core Hummingbird processor and 512MB of DRAM. The servers were all connected to one top-of-rack switch. The devices were located on the same local area network as the servers, and communicated with the server either through a wireless connection or T-mobile 3G links.

8.3 Microbenchmarks

We measured the DK for latency and throughput using closed-loop RPCs. Latencies were measured at the client. Before taking measurements, we first sent several thousand requests to warm up the JVM to avoid the effects of JIT and buffering optimizations.

RPC Latency Comparison. We compared the performance of Sapphire RPC to Java RMI and to two widely used communication models: Thrift and REST. Apache Thrift [3] is an open-source RPC library used by Facebook, Cloudera and Evernote. REST [25] is a popular low-level communication protocol for the Web; many sites have a public REST API, including Facebook and Twitter. We measured REST using a Java client running the standard `HttpURLConnection` class and a PHP script running on Apache 2.2 for method dispatch.

Table 5 shows request/response latencies for intra-node (local), server-to-server, tablet-to-server, server-to-tablet and tablet-to-tablet communications on null requests for all four systems. While Thrift was slightly faster in all cases, Java RMI and Sapphire were comparable and were both faster than the Java REST library.

Sapphire uses Java RMI for communication between DK servers; however, we dispatch method calls to SOs through the DK. This additional dispatch caused the latency difference between Java RMI and Sapphire RPC. The extra cost was primarily due to instantiating and serializing Sapphire's RPC data object (which is not required for a null Java RMI RPC). We could reduce this cost by using a more efficient RPC and serialization infrastructure, such as Thrift.

Note that even without optimization, Sapphire was faster than REST, which is probably the most widely used communication framework today. Furthermore, we could

Table 5: Request latencies (ms) for local, server-to-server, tablet-to-server, server-to-tablet and tablet-to-tablet. Note that REST does not support communication to tablets.

RPC Protocol	Local	S→S	T→S	S→T	T→T
Sapphire	0.08	0.16	5.9	3.4	12.0
Java RMI	0.05	0.12	4.6	2.0	7.2
Thrift	0.04	0.11	2.0	2.0	3.6
REST	0.49	0.64	7.9	-	-

not show REST performance for server-to-tablet and tablet-to-tablet because REST's client-server architecture cannot accept HTTP requests on the tablet. Thus, REST can be used only for tablet-to-server communication, requiring the application to explicitly manage communication forms such as server-to-client or client-to-client.

Throughput Comparison. We measured request throughput for the Sapphire DK and Java RMI. The results (Figure 7) showed similar throughput curves, with Java RMI object throughput approximately 15% higher than that for Sapphire Objects. This is because Sapphire null RPCs are not truly empty: they carry a serialized structure telling the DK how to direct the call. To break the cost down further, we measured the throughput of a Java RMI carrying a payload identical to that of the Sapphire null RPC. This reduced the throughput difference to 3.6%; this 3.6% is the additional cost of Sapphire's RPC dispatching in the DK, with the remainder due to the cost of serialization for the dispatching structure. Again, there are many ways to reduce the cost of this communication in Sapphire, but we leave those optimization to future work.

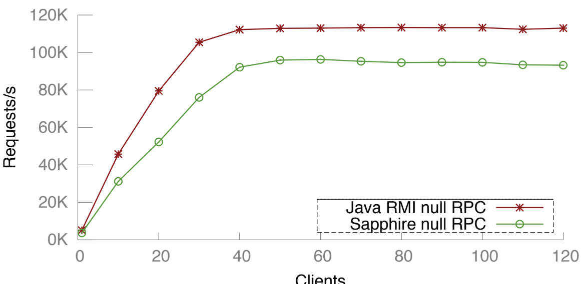


Figure 7: Throughput of a Sapphire Object versus an RMI Object.

Sapphire DK Operation Cost. We measured the latency of several DK services. DK call latency depends on the size and complexity of the object, since we use Java serialization. Table 6 shows latency results for creating, replicating, and moving SOs on servers and tablets. Operation latencies were low when executed on cloud servers. Tablets were considerably slower than cloud servers. However, we expect most management operations such as these to be performed in the cloud (i.e., we do not expect tablets to create large numbers of SOs).

The SO instantiation process can be expensive because

Table 6: Sapphire DK API latencies (ms).

Object	create		replicate		move (over WiFi)			
	S	T	S	T	S→S	T→S	S→T	T→T
Table	1.1	28	0.5	15	1.9	42	16	66
Game	1.1	29	0.5	16	2.1	49	19	67
TableMgr	1.1	27	0.6	18	2.2	50	16	78

the DK must create several objects locally: the SO, the SO stub, the DM Proxy and the DM Instance Manager. The DK must also create the DM Coordinator remotely on a DK-FT node and register the SO with the OTS. Communication with the DK-FT node and the OTS accounted for nearly half the instantiation latency.

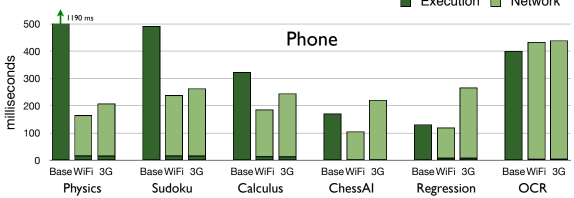
8.4 Deployment Manager Performance

We measured the performance of five categories of DMs: caching, replication, peer-to-peer, mobility, and scalability. Our goal was to examine their effectiveness as extensions to the DK and the costs and trade-offs of employing different DMs.

Caching. We evaluated two caching DMs: LeaseCaching and ConsistentCaching. As expected, caching significantly improved the latency of reads in both cases. For the TodoList SO, which uses the LeaseCaching DM, caching reduced read latency from 6 ms to 0.5 ms, while write latency increased from 6.1 ms to 7.5 ms. For the Game SO, which uses the ConsistentCaching DM, all read latencies decreased, from 7-13 ms to 2-3 ms. With consistent caching, the write cost to keep the caches and cloud synchronized was significant, increasing from 29 ms to 77 ms. Overhead introduced by the DM was due to the use of serialization to determine read vs. write operations. For writes, the whole object was sent to be synchronized with the cloud, instead of a compact patch.

Code offloading. We measured our ported, compute-intensive applications with the CodeOffloading DM for the Nexus 7 tablet and the Galaxy S smartphone. Figure 8 shows the latencies for running each application locally on the device (shown as Base), offloaded to the cloud over WiFi, and offloaded over 3G. The offloading trade-offs varied widely across the two platforms due to differences in CPU speed, wireless, and cellular network card performance. For example, for the Calculus application, cloud offloading was better for the phone over both wireless and 3G; however, for the tablet it was better only over wireless. For the Physics engine, offloading was universally better, but it was particularly significant for the mobile device, which was not able to provide real-time simulation without code offloading.

These cross-platform differences in performance show the importance of flexibility. An automated algorithm cannot always predict when to offload and can be costly. Therefore, it is important for the programmer to be able to easily change deployment to adapt to new technologies.



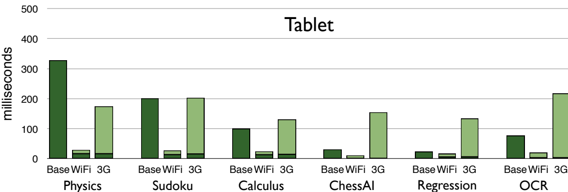


Figure 8: Code offloading performance.

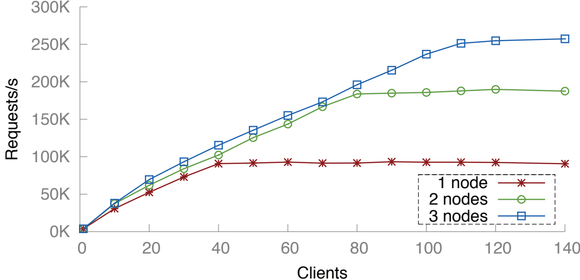


Figure 9: Effects of applying the LoadBalancedFrontEnd DM.

Scalability. We built the LoadBalancedFrontEnd DM to scale a stateless SO under heavy load. The DM creates a given number of non-consistent replicas of an SO and assigns clients to the replicas in a round-robin fashion. Figure 9 shows the throughput of the SO serving null RPCs when the DM creates up to 3 replicas. Throughput scaled linearly with the number of replicas until the network saturated at 257,365 requests/second.

Peer-to-Peer Deployments. Sapphire lets programmers move objects easily between clients and servers, enabling P2P deployments that would be difficult or impossible in existing systems. We measured three deployments for the Game SO from our multi-player game: (1) without a DM, which caused Sapphire to deploy the SO on the server where it is created; (2) with the KeepOnDevice DM, which dynamically moved the Game object to a device that accessed it; and (3) with the ConsensusRSM-P2P DM, which created synchronized replicas of the Game SO on the callers' devices.

For each deployment, Figure 10 shows the latency of the game's read methods (`getScrambleLetters()`, `getPlayerTurn()` and `getLastRoundStats()`) and write methods (`play()` and `pass()`). With the Game SO in the cloud, read and write latencies were high for both players. With the KeepOnDevice DM, the read and write latencies were extremely low for the device hosting the SO, but somewhat higher for the other player, compared to the cloud version. Finally, with the ConsensusRSM-P2P DM, read latencies were much

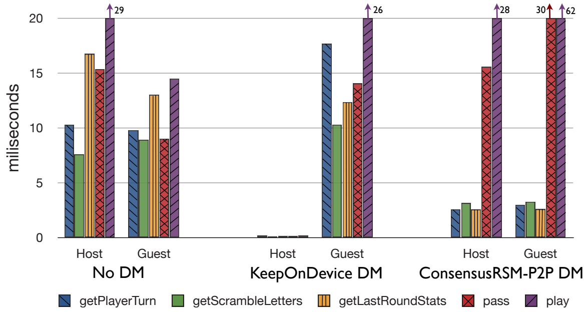


Figure 10: Multi-player Game: different deployment schemes.

lower for both devices, while write latencies were higher. In our scenario, the two tablets and the server were on the same network. In cases where the two players are close on the network and far from the server, the peer-to-peer DMs would provide a valuable deployment option.

With the DMs, no cloud servers were needed to support the Game SO; this reduced server load, but Game SOs were no longer available if the hosting device were disconnected. This experiment shows the impact of different deployment options and the benefit of being able to flexibly choose alternative deployments to trade off application performance, availability, and server load.

9 Related Work

Researchers have built many systems to help applications cope with deployment issues. Code-offloading systems, like COMET [30], MAUI [19], and CloneCloud [14], automatically offload computationally intensive tasks from mobile devices to cloud servers. Distributed storage systems [21, 12, 16] are a popular solution for server-side scalability, durability and fault-tolerance. Systems like PADS [7], PRACTI [6] and WheelFS [65] explored configurable deployment of application *data* but not runtime management of the entire application. Systems like Bayou [67], Cimbiosys [55] and Simba [2] offer client-side caching and offline access for weakly connected environments. Each of these systems only solves a subset of the deployment challenges that mobile/cloud applications face. Sapphire is the first distributed system to provide a unified solution to deployment for mobile/cloud applications.

When building Sapphire's DM library, we drew inspiration from existing mobile/cloud deployment systems, including those providing: wide-area communication [34], load-balancing [31, 72], geographic replication [43, 63], consensus protocols [37, 52], and DHTs [64, 57, 45].

Similar to our goal with Sapphire, previous language and compiler systems have tried to unify the distributed environment. However, unlike Sapphire, these solutions have no flexibility. They either make all deployment decisions for the application – an approach that doesn't work for the wide range of mobile/cloud requirements – or they leave all deployment up to the programmer. Compilers like Coign [32], Links [15], Swift [13] and Hop [60] automatically partition applications, but give programmers no control over performance trade-offs. Single language domains like Node.js [51] and Google Web Toolkit [29] create a uniform programming language across browsers and servers, but leave deployment up to the application. For mobile devices, MobileHTML5 [47], MobiRuby [48] and Corona [17] support a single cross-platform language. Sapphire supports a more complete cross-platform environment, but programmers can select deployments from an extensive (and extensible) library.

The DK's single address space and distributed object model are related to early distributed programming systems such as Argus [41], Amoeba [66] and Emerald [35]. Modern systems like Orleans [10] and Tango [4] provide cloud- or server-side services. Fabric [42] extends the work in this space with language abstractions that provide security guarantees. These systems were intended for homogeneous, local-area networks, so do not have the customizability and extensibility of the Sapphire DK.

Overall, existing or early distributed programming systems are not *general-purpose, flexible* or *extensible* enough to support mobile/cloud application requirements. Therefore, in designing Sapphire, we drew inspiration from work that has explored customizability and extensibility in other contexts: operating systems [24, 8, 26, 59, 40], distributed storage [7, 20, 65, 61, 27], databases [11, 5], and routers and switches [36, 46].

10 Conclusion

This paper presented Sapphire, a system that simplifies the development of mobile/cloud applications. Sapphire's Deployment Kernel creates an integrated environment with location-independent communication across mobile devices and clouds. Its novel deployment layer contains a library of Deployment Managers that handle application-specific distribution issues, such as load-scaling, replication, and caching. Our experience shows that Sapphire: (1) greatly eases the programming of heterogeneous, distributed cloud/mobile applications, (2) provides great flexibility in choosing and changing deployment decisions, and (3) gives programmers fine-grained control over performance, availability, and scalability.

Acknowledgements

This work was supported by the National Science Foundation (grants CNS-0963754, CNS-101647, CSR-1217597), an NSF Graduate Fellowship, the ARCS Foundation, an IBM PhD Scholarship, Google, and the Wissner-Slivka Chair in Computer Science & Engineering. We thank our shepherd Doug Terry and the reviewers for their helpful comments on the paper. Finally, we'd like to thank the UW Systems lab for their support and feedback throughout the project.

References

[1] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: A client notification service for internet-scale applications. In *Proc. of SOSP*, 2011.

[2] N. Agrawal, A. Aranya, and C. Ungureanu. Mobile data sync in a blink. In *Proc. of HotStorage*, 2013.

[3] Apache. Apache Thrift, 2013. http://thrift.apache.org.

[4] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proc. of SOSP*, 2013.

[5] D. Batoory, J. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An extensible database management system. *IEEE Transactions on Software Engineering*, 1988.

[6] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *Proc. of NSDI*, 2006.

[7] N. M. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm. PADS: A policy architecture for distributed storage systems. In *Proc. of NSDI*, 2009.

[8] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proc. of SOSP*, 1995.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.

[10] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *Proc. of SOCC*, 2011.

[11] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. *Object and file management in the EXODUS extensible database system*. Computer Sciences Department, University of Wisconsin, 1986.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 2008.

[13] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. of SOSP*, 2007.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proc. of EuroSys*, 2011.

[15] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. of FMCO*, 2006.

[16] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. of OSDI*, 2012.

[17] Corona SDK, 2013. http://www.coronalabs.com/.

[18] J. Cowling, D. R. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: Location-aware membership management for large-scale distributed systems. *Proc. of USENIX ATC*, 2009.

[19] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *Proc. of MobiSys*, 2010.

[20] M. Dahlin, L. Gao, A. Nayate, A. Venkataramana, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. of NSDI*, 2006.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, 2007.

[22] D. Diephouse and P. Brown. Building a highly scalable, open source Twitter clone, 2009. http://fr.slideshare.net/multifariousprb/building-a-highly-scalable-open-source-twitter-clone.

[23] Dropbox, 2013. http://dropbox.com.

[24] D. R. Engler, M. F. Kaashoek, et al. Exokernel: An operating system architecture for application-level resource management. In *Proc. of SOSP*, 1995.

[25] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[26] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. of SOSP*, 1997.

[27] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key-value store. In *Proc. of OSDI*, 2010.

[28] 2013. https://developers.google.com/google-apps/marketplace/sso.

[29] Google web toolkit. https://developers.google.com/web-toolkit/, October 2012.

[30] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code offload by migrating execution transparently. In *Proc. of OSDI*, 2012.

[31] HAProxy: A reliable, high-performance TCP/HTTP load balancer, 2013. http://haproxy.1wt.eu/.

[32] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *Proc. of OSDI*, 1999.

[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC*, 2010.

[34] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: a toolkit for mobile information access. In *Proc. of SOSP*, 1995.

[35] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. In *Proc. of SOSP*, 1987.

[36] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proc. of SOSP*, 1999.

[37] L. Lamport. Paxos made simple. *ACM Sigact News*, 2001.

[38] C. Leau. Spring Data Redis - Retwis-J, 2013. http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[39] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proc. of SOSP*, 2011.

[40] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proc. of SOSP*, 1975.

[41] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proc. of SOSP*, 1987.

[42] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. of SOSP*, 2009.

[43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. of SOSP*, 2011.

[44] J. Maassen, R. Van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 2001.

[45] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of IPTPS*, 2002.

[46] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[47] Moblie HTML5, 2013. http://mobilehtml5.org.

[48] MobiRuby, 2013. http://mobiruby.org/.

[49] MySQL, 2013. http://www.mysql.com/.

[50] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Proc. of Java Grande*, 1999.

[51] Node.js, 2013. http://nodejs.org/.

[52] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[53] Parse, 2013. http://parse.com.

[54] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 2000.

[55] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proc. of NSDI*, 2009.

[56] Redis: Open source data structure server, 2013. http://redis.io/.

[57] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of Middleware*, 2001.

[58] Amazon S3, 2013. http://aws.amazon.com/s3/.

[59] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of OSDI*, 1996.

[60] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Companion*, 2006.

[61] A. Siegel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. In *Proc. of the Workshop on the Management of Replicated Data*, 1990.

[62] Simple object access protocol. http://www.w3.org/TR/soap/.

[63] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. of SOSP*, 2011.

[64] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM*, 2001.

[65] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proc. of NSDI*, 2009.

[66] A. S. Tanenbaum, R. Van Renesse, H. Van Staveren, G. J. Sharp, and S. J. Mullender. Experiences with the Amoeba distributed operating system. *Commun. ACM*, 1990.

[67] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP*, 1995.

[68] Twimight open-source Twitter client for Android, 2013. http://code.google.com/p/twimight/.

[69] Voldemort: A distributed database, 2013. http://www.project-voldemort.com/voldemort/.

[70] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 1992.

[71] D. A. Wheeler. SLOCCount, 2013. http://www.dwheeler.com/sloccount/.

[72] Zen load balancer, 2013. http://www.zenloadbalancer.com/.

Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems

Riley Spahn, Jonathan Bell, Michael Z. Lee*, Sravan Bhamidipati,
Roxana Geambasu, *and* Gail Kaiser
*Columbia University, *The University of Texas at Austin*

Abstract

Support for fine-grained data management has all but disappeared from modern operating systems such as Android and iOS. Instead, we must rely on each individual application to manage our data properly – e.g., to delete our emails, documents, and photos in full upon request; to not collect more data than required for its function; and to back up our data to reliable backends. Yet, research studies and media articles constantly remind us of the poor data management practices applied by our applications. We have developed Pebbles, a fine-grained data management system that enables management at a powerful new level of abstraction: *application-level data objects*, such as emails, documents, notes, notebooks, bank accounts, etc. The key contribution is Pebbles's ability to discover such high-level objects in arbitrary applications *without* requiring any input from or modifications to these applications. Intuitively, it seems impossible for an OS-level service to understand object structures in unmodified applications, however we observe that the high-level storage abstractions embedded in modern OSes – relational databases and object-relational mappers – bear significant structural information that makes object recognition possible and accurate.

1 Introduction

Despite recent high-profile failures in applications' management of our data [2], in the absence of system-level support for fine-grained data organization, we are forced to entrust them with our data. When users perform day-to-day data management activities – deleting individual emails, identifying specific data that was viewed, or sharing pictures – they are forced to rely on applications to behave properly. Yet, a 2010 study of 30 popular Android applications showed that 20 leaked sensitive data, such as contacts or locations [11]. Our own study of deletion practices within mobile apps, described later in this paper, revealed that 18 of 50 popular Android applications left information behind instead of deleting it. Notably, we found that until 2011, Android's default email application left behind the attachments of deleted emails while deleting the messages themselves.

Although a plethora of system-level data management tools exist – including encrypted file systems [14, 16], deniable file systems [42], auditing file systems [12], or assured delete systems [28] – these tools operate at a single level of abstraction: *files*. Without a one-to-one mapping between user-relevant objects (for example, individual email messages in a mail client or documents in a word processor) and files, such systems provide poor granularity, preventing end-users from protecting individual objects that matter to them.

Consider Android's default email application: it stores each email's contents and to/from/subject fields as several rows in a SQLite database (all emails are stored in the same DB, which is itself stored as a single file), attachments as files, and cached renderings of messages in different files. Such complex object-to-file mappings are typical in Android, as our large-scale measurement study of Android storage patterns shows (§3). Moreover, others have observed complex storage layouts in other OSes, such as OSX, where researchers have concluded that "a file is not a file" but a complex structure with complex access patterns [18].

Given the complexity of these object-to-file mappings, we ask: is it possible for system-level tools to support management and protection at the granularity of user-relevant objects? Intuitively, this would require developers to specify the structure of their applications' persisted data to the operating system. Nevertheless, we observe that the high level storage abstractions included and predominant in today's operating systems – the SQLite relational database in Android and the CoreData object-relational mapper in iOS – bear sufficient structural information to recover these user-relevant data objects from unmodified applications.

We call these objects *logical data objects* (LDO), examples of which include an email (including its to, from, subject, body, attachments and any other related information); a mailbox including all emails in it; a bank account in a personal finance application; etc. We present *Pebbles*, a system that exposes LDOs to protection tools, without introducing any new programming models or interfaces, which can be prone to programmer error, slow adoption, or incompatibility with legacy applications.

We implemented Pebbles and several new protection tools based on it on the Android platform. Each of these tools provides protection at the LDO level, leveraging Pebbles to greatly simplify their development. Using Pebbles tools, users can mark objects from their existing applications to verify their proper deletion, protect their access from other applications, and back them up to the clouds they trust.

In a study of 50 popular Android applications, we found Pebbles to be highly effective in automatically identifying LDOs. Across these apps, object recognition recall was 91% and precision was 97%. In other words, in 91% of the cases, there was no leakage of data from user-visible objects to LDOs, and in 97% of the cases, there was no over-inclusion of extra data beyond user expectation in LDOs. Pebbles relies on several key assumptions based on common practices. Many of the cases in which Pebbles had poor accuracy, it could have been addressed had the developers followed these common practices.

Overall, this work makes the following contributions:

1. A study of over 470,000 Android apps, analyzing, for the first time at scale, the storage abstractions in common use today (§3). Our results suggest major differences compared to traditional storage abstractions, which render file-level data management ineffective while creating untapped opportunities for object-level data management.
2. The first design and implementation of a persistent data object recognition system that requires no app changes (§4 and §6). Our design taps into the opportunities observed from our large-scale Android app study. We make our code available from `https://systems.cs.columbia.edu/projects/os-abstractions`.
3. Four protection tools implemented atop Pebbles, demonstrating the power and value of application-level objects to protection tools (§5).
4. An evaluation of LDO construction accuracy with Pebbles over 50 popular applications from Google Play, showing it to be effective in practice (§7) and underscoring its well-defined failure modes (§8).

2 Motivation and Goals

We begin by presenting a set of example scenarios that highlight the need for fine-grained data management support within modern OSes.

2.1 Example Scenarios

Scenario 1: Object Deletion: Ann, an investigative journalist, has received an extremely sensitive email on her phone with an attachment that identifies her sources. To protect her sources, Ann does her due diligence by deleting the email immediately after reading its contents and restarting her phone to clean up any traces left in memory. Her phone is already configured with an assured-delete file system [28] that deletes data promptly upon request. Worried that the application might have created a copy of her data without her knowledge or control, she wonders: *Is there any remnant of that email left anywhere on the phone?* She is disappointed to realize that she has zero visibility into the data stored on her device. Weeks later, she learns that her fears were well-founded: the email app she is using contains a bug that leaves attachments intact when an email is deleted.

Scenario 2: Object Access Auditing: Bob, a financial auditor, uses his phone for all interactions with client data while on field engagements. Recently, Bob's device was stolen. Fearing that his fingerprint unlock might not withstand motivated attackers [41], Bob asked his IT admin a natural question: *Has any of my clients' data been exposed?* The admin's answer was mixed. Although activity on Bob's phone was tracked by a remote auditing file system [12], the logs show that a file, `/data/data/com.android.email/cache/7dcee8`, was accessed immediately before the phone's wipe-out. The file stores the HTML rendering of an email, but no one knows *which* email. Bob is left wondering what he should disclose to clients about the potential exposure of their data, and to *which* clients, since neither he nor the IT staff can map that file to a specific client or email.

Scenario 3: Object Access Restriction: Carla, a local politician, uses her phone to take photos for professional purposes, but she has several personal photos on it as well. She uses a cloud-based photo editor to enhance her promotional photos before posting them. Due to the coarse-grained permissions model of her Android device, she must provide this photo editor with access to all of her photos in order to use it. Carla is concerned that the photo editor may be secretly collecting all the photos from her device, including several potentially sensitive photos that could be politically compromising.

2.2 Goals and Assumptions

The above hypothetical users, along with millions of real-life users of mobile technology, have a mental model of application-level objects that is not matched by current protection tools. Ann wants to ensure that a particularly sensitive email is deleted in full, including attachments, to, from, any related caches, and other fields; Bob wants to know the sender or contents of a compromised email instead of a meaningless file name; Carla wants to protect a few of her most sensitive photos from prying applications. Traditional protection tools, such as file-based encryption, auditing, or secure deletion cannot fulfill these needs because the mapping between objects and files is application-specific and complex. The alternative, whole-disk encryption [1, 38], does not provide the flexibility that these users need.

To support such object-level data management needs, we developed Pebbles, a system that automatically reconstructs application-level logical data objects (LDOs) from unmodified applications. Pebbles exposes these LDOs to any system-wide protection tool that could benefit from understanding application-level objects. An encryption system could use LDOs to support meaningful fine-grained protection as an extra layer on top of whole-

disk encryption. An auditing system could use LDOs to provide meaningful information about an accessed component. An object manager could reveal to users which parts of an object are left after deletion. And a backup system could let users choose their most sensitive objects for backup onto a trusted, self-managed server, letting the rest be backed up into the cloud.

Goals. The Pebbles design was guided by three goals:

G1: Accurate and Precise Object Recognition: Pebbles objects (LDOs) must closely match application-level persisted objects. This includes: (a) *avoiding data leaks* (if an item belongs to an LDO it must be included), and (b) *avoiding data over-inclusions* (if an item does not belong to an LDO it should not be included).

G2: Meaningful Granularity: Pebbles must recognize LDOs that are meaningful to users, such as individual emails.

G3: No New Application APIs: Pebbles must not require app developers to use new APIs; it can recommend developers to follow existing common practices but must work well even if they do not precisely follow.

Our first goal is accurate and precise object recognition (*G1*). We aim to achieve (1) good object recognition *recall* by avoiding leaks and (2) good object recognition *precision* by avoiding over-inclusions. We acknowledge that perfect recall or precision cannot be guaranteed in either an unsupervised approach or in a supervised API approach with imperfect developers, since a poorly written app could convolute data structure in a way that Pebbles cannot recover. However, we wish to formulate clearly all potential sources of leakage, to design mechanisms to address the leakages for most applications (§4.2), and to remind developers how they could avoid such leakages by following existing common practices (§8).

Related to G1, our second goal (*G2*) is to recognize relevant and meaningful LDOs. For example, in an email app, Pebbles should be able to recognize individual emails, not just coarse accounts with many emails. We note here that Pebbles identifies application-level objects that are persisted in stable storage, and we assume that those have a direct mapping onto the objects that users interact with and wish to protect.

G3 stems from our skepticism that developers will convert applications to use new security-related APIs or correctly use such APIs. However, we do expect that most developers will follow certain common practices (as evaluated in §3). Pebbles addresses this by leveraging application-level semantics already available within storage abstractions such as database schemas, XML structures, and the file system hierarchy. Pebbles also provides recommendations for developers which are rooted in already popular development practices (§8).

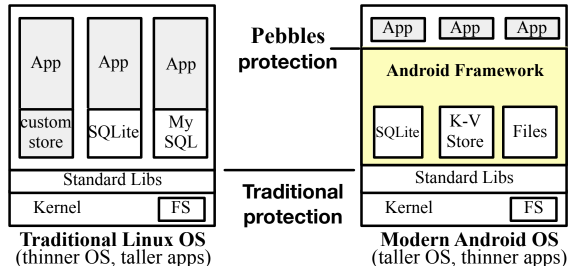


Fig. 1: **OS Storage Abstraction Evolution.** Modern OSes provide higher-level abstractions for data management, yet protection is often at the traditional file level. Pebbles, our aligns data protection with modern abstractions.

Threat Models and Assumptions. Pebbles is designed to support fine-grained data management – such as encryption, auditing, and deletion of individual emails, photos, or documents – within modern OSes. The specific threat model for a given protection tool depends on that tool's goal; however, Pebbles's mechanisms should bolster the guarantees applications can provide. In general, we assume that protection tools are *trusted* system-wide services. This is similar to assumptions made by encrypted file systems, assured-delete file systems, and other current fine-grained data management tools.

We also assume that mobile applications that create or have access to a particular object, or part thereof, will not obfuscate their data's structure or act maliciously against Pebbles. For example, they will not create their own data formats and will not willfully interfere with analysis mechanisms involved in object discovery. An application that has not yet been given access to data of a particular object, however, need not be trusted.

The scope of Pebbles is confined to those application-level objects that are persisted into a device's stable storage. We explicitly ignore attackers with access to either RAM or the underlying OS or hardware. If volatile memory protection is important, we recommend combining Pebbles with secure memory deallocation [6, 7, 15], OS buffer cleaning [10], and idle in-RAM data eviction [39] mechanisms. We also assume that secure disk scrubbing [29, 40] is deployed. In addition, while many modern applications include a cloud component, which stores or backs up data, Pebbles currently ignores that component. In the future, we plan to extend Pebbles LDOs to transcend the local and cloud environments.

While some may believe that users are incapable of dealing with fine-grained controls, we believe that there are many circumstances in which users want and are capable of handling *some* level of control, particularly for their most sensitive data. Evidence that users are capable of handling, and require, some level of control *when they feel it is important for them to do so* is available in prior studies [5, 20]. Such evidence can also be gauged

Storage Abstraction	# Apps (of 98)	Example Apps
No storage	5	Cardio Trainer
DB only	43	**CWMoney**, Amazon, BestBuy, Browser, Calendar, Contacts, ColorNotes, EverNote
FS only	3	Exchange Rates
KV only	5	Google Talk, Biorythms
DB+FS	24	**OINote**, Angry Birds, DropBox, Gallery
DB+KV	1	Twitter
FS+KV	2	Adobe Reader, Temple Run
DB+FS+KV	15	**Email**, Antivirus, Amazon Kindle, Astro File Manager, Box, EBay

(a) Use of SQLite (DB), FS, and key/value (KV) store

Storage Library	# of Apps (of 476,375)
ORMLite	6,846 (1.4%)
SQLCipher	168 (0.3%)
DB4o	116 (0.2%)
H2	16 (0.0%)
Other 4 libs combined	38 (0.0%)

(b) Third-party library use

App	Object	DB/FS Use
Email (DB+FS+KV)	Email	to/from/date in one DB table; contents in another table; attachments in FS
	Mailbox	name/server/account in one DB table; includes emails; backup in kv
	Account	address/meta data in one DB table; includes mailboxes, emails
OINote (DB+FS)	Note	title/note/tags/ in one DB table; notes exported as files in /sdcard FS
CWMoney (DB only)	Expense	name/amount in one DB table
	Category	category name in one DB table; includes expenses
	Account	name/balance in one DB table; includes categories, expenses

(c) Example object structures

Fig. 2: **Storage API Usage in 98 Android Applications.** (a) Number of apps that use the various storage abstractions in Android. Most apps use DB, but many also use FS and KV together with DB. (b) Use of eight other storage libraries among 476K free apps from Google Play. Third-party storage libraries are largely irrelevant. (c) Structure of sample objects in a few popular apps. Object structure is complex and spans multiple abstractions.

from the immense popularity of data hiding apps, such as Vault-Hide [25] and KeepSafe Vault [19], which have garnered over 10 million downloads each and let users hide data, such as photos, contacts, and SMSes.

3 Study: Android Storage Abstractions

The Pebbles design is motivated and informed by our high-level observation that storage abstractions within modern OSes are evolving in major yet unquantified ways. Fig.1 shows this evolution. Specifically, we hypothesize that the inclusion of high-level storage abstractions, such as the SQLite database in Android or the CoreData abstraction in iOS, has created a new "narrow waist" for storage abstractions that largely hides the traditional hierarchical file system abstraction. These new storage abstractions should bear sufficient structure to let us reverse engineer application-level data objects from the OS's vantage point.

In this section, we perform a simple measurement study to gauge the use of these abstractions and extract useful insights to inform our design of Pebbles. We specifically ask the following questions:

Q1 What storage abstractions do Android apps use?
Q2 How do individual apps organize their data?
Q3 How are these abstractions used?

Background. Android provides three storage abstractions [13] relevant to this paper: 1. *SQLite Database:* Stores structured data. 2. *XML-based Key/Value Store:* Stores primitive data in key/value pairs (also known as the SharedPreferences API). 3. *Files:* Stores unstructured data on the device's flash memory.

Methodology. We ran both *static* and *dynamic* experiments. Static experiments can be run at large scale but lack precision, while dynamic experiments provide precise answers but can only be run at small scale. For static experiments, we decompiled Android applications and searched their source code for imports of the storage abstractions' packages (e.g., `android.database.sqlite`). We ran large-scale, static experiments on 476,375 apps downloaded through a February 2013 crawl of Google Play [44], the main Android app market. For the dynamic experiments (over 98 apps), we installed Android apps on a Nexus S phone, manually interacted with them, and logged their accesses to the various APIs. These were some of the most popular apps, cutting across categories such as email clients, editors, banking, shopping, social, and gaming.

Results. *Q1 Answer: Apps primarily use SQLite, but use other abstractions as well.* Fig. 2(a) classifies apps according to the Android-embedded storage abstractions they use during execution. It shows that the usage of Android-provided abstractions – SQLite (denoted DB) and the key/value store (denoted KV) – eclipses the traditional file abstractions (denoted FS). Very few apps rely on the FS as their only storage abstraction (4/98). Almost half of the apps rely solely on SQLite for all of their storage needs (43/98), while almost all apps that have some local storage use SQLite (81/92). Even apps that one would consider to be primarily file-oriented (e.g., Astro File Manager, DropBox) use SQLite. A significant fraction of the apps (41/98) rely on more than one abstraction, and a notable fraction (15/98) rely on all three abstractions. This last result suggests a complex disk layout, a topic discussed further below. Overall, the most popular formations are: DB-only (43/98), DB+FS (23/98), and DB+FS+KV (15/98).

A related question is whether mobile apps use storage abstractions *other* than those provided by Android. An-

gry Birds, for example, stores game data and high scores in opaque binary files. We also searched the Internet for recommended Android storage options beyond those included in the OS, finding eight third-party libraries. We searched our 476K-app corpus for use of those libraries, and present the results in Fig. 2(b). None of these libraries are popular: only 2% of the apps use even one of them. Our dynamic experiments found that none of these libraries are used and provided no indication of additional libraries that we might have overlooked.

Q2 Answer: Data objects span multiple storage abstractions. Fig. 2(c) shows the structures of several logical data objects, representative of what users think and care about in various applications. It shows that objects often have complex structures that involve multiple storage abstractions. For example, Android's default email client, an example of the DB+FS+KV formation, stores various fields of the email object in two DB tables, attachments in the FS, and account recovery information in the KV. Object structure is fairly complex even for DB-only apps, such as CWMoney, a personal finance app, where a category includes metadata in one table and all expenses in another table. It thus spans multiple tables that are not linked together through explicit foreign keys. This suggests that protecting each storage abstraction separately will not work: any data protection abstraction at the end-user object level must span multiple storage abstractions.

Q3 Answer: SQLite is the hub for data management. Given this complexity, a natural question concerns how one can even begin to build some meaningful protection abstraction. Using a modified TaintDroid (a popular data flow taint tracking system for Android [11]) version, we tracked the flow of data between storage abstractions, confirming that at least 70/81 apps that use the DB use it as a *central hub* for managing their data. By central hub, we mean that data flows mostly from the DB into the FS/KV (when they are used) or is accessed using pointers from the DB; an observation that was true for 27 of the 38 apps that use FS or KV in addition to the DB. For example, many apps, including Email, use files to store caches of rendered versions of data stored in SQLite (such as the body of an email) or blobs of data that are indexed and managed through SQLite (such as the contents of pictures, videos, or email attachments).

Thus, SQLite is not just frequently used; it is the central abstraction in Android that originates or indexes much of the data stored in the other abstractions. This result is encouraging because, intuitively, relational databases bear more explicit structure.

Implications for the Pebbles Design. Overall, our results suggest that while the storage abstraction landscape is fairly complex in Android, there is sufficient uniformity to warrant constructing of a broadly applicable object system. Such a system must detect relationships between objects stored in different abstractions. The results suggest that SQLite, a relational database that bears significant inherent structure, is the predominant storage abstraction in Android. Raw files, which lack such structure, are just used for overflow storage of bulk data, such as images, videos, and attachments. Based on these insights, we construct Pebbles, the first system to recognize application-level objects within modern operating systems without application modifications.

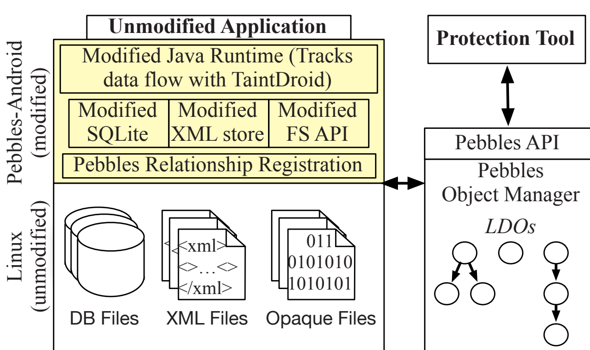


Fig. 3: **The Pebbles Architecture.** Consists of a modified Android framework and a device-wide Pebbles Object Manager. The modified framework identifies relationships between persisted data items, such as rows, XML elements, or files. The Pebbles Object Manager uses those relationships to construct an object graph; nodes map to persisted data items and edges map to relationships.

4 The Pebbles Architecture

Pebbles aims to reconstruct application-level LDOs – emails and mailboxes in an email app, saved high scores in a game, etc. – from the bits and pieces stored across the various data storage abstractions without requiring application modifications.

4.1 Overview

Fig. 3 shows the Pebbles architecture, which consists of two core components: (1) *Pebbles Android*, a modified Android framework that interposes on the various storage APIs, and (2) the *Pebbles Object Manager*, a separate device-wide entity for building object graphs and interacting with protection tools.

At the most basic level, the Pebbles Android framework understands units of storage (e.g., rows in DB, elements in XML, and files in FS) which become nodes in our object graph. The Pebbles Android framework then retrieves explicit relationships between these nodes and derives implicit relationships by tracking data flows between these units. The Pebbles Android framework registers these relationships with the Pebbles Object Manager using an internal registration API. The Pebbles Object Manager then stores these relationships, compiles a device-wide *object graph*, derives LDOs from the graph, and exports the LDOs to protection tools via the Pebbles

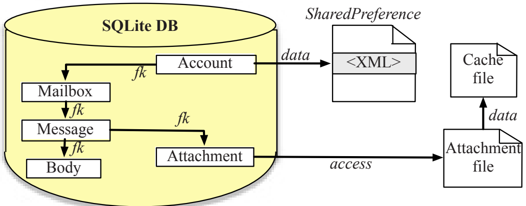


Fig. 4: **Android Email App Object Structure.** A simplified object graph for one account with one mailbox, message, and attachment. Each node represents an individual file, row, or XML element, and each edge represents a relationship. While objects can be spread across the DB, FS, and Shared Preferences, the DB remains the hub for all data.

API. LDOs are defined as follows: given a node in the graph (e.g., corresponding to a row in the `Email` table) an LDO is the transitive closure of the nodes connected to it. §7 evaluates Pebbles performance in terms of precision and recall. In the context of the graph, a failing of recall is missing nodes which should be included in a transitive closure ("leakage"); a failing of precision is including nodes which should *not* be included in a transitive closure ("over inclusion").

To provide a concrete example of the challenges faced by Pebbles, consider Fig. 4, a simplified view of how data is stored by the default Android Email application. As described previously in §3, this app stores its data across all three storage abstractions: SQLite database, SharedPreference and individual files. Although a SharedPreference is used for account recovery, and several files are used to store an attachment and a cached rendering of it, the majority of the data is stored in SQLite.

4.2 Building the Object Graph

The object graph is the center of innovation in Pebbles: it directly represents Pebbles's understanding of the structure of an app's data and lets it construct LDOs. Each file, row, and XML element is assigned a 32 bit device-wide globally-unique ID (GUID) that is stored with the data item, which are hidden from and unmodifiable by applications. For database rows, the GUID is stored as an extra column in the row's table; for XML, it is stored as an attribute of each element; and for files, it is stored in an extended attribute. When a row, element, or file is read, the data coming from it is "tainted" with its GUID and tracked in memory using a modified version of the TaintDroid taint tracking system [11].

Pebbles builds the object graph incrementally by adding new files/rows/XML elements as nodes into the graph as they are created. It also adds directed edges (called *relationships*) between nodes in the graph as they are discovered. For example, when data tainted with one GUID is written into a file/row/XML element with another GUID, a relationship is registered. All nodes and edges of the graph are registered by the modified Android framework with the Pebbles Object Manager, where they are persisted in a database. We next describe the mechanisms used to build this graph, formalized in Fig. 5.

Data flow propagation relationships: It is easy to see a strawman approach to detecting relationships between objects: when Pebbles detects that data tainted with node A's GUID is written into node B, it adds $A \leftrightarrow B$ to the object graph. This approach can capture all data flow relationships that occur within an application, regardless of the storage abstraction used. However, without precise information about the relationship between the two nodes, Pebbles is forced to assume the "worst case" scenario: that both nodes are part of the same LDO. Left unchecked, this so called taint explosion could eventually lead to all of an app's objects being included in the same LDO. Such behavior contradicts our primary goal of accurate and precise object recognition (G1). As we will see in §7.1, this naïve approach leads to unacceptably low precision (70%).

Utilizing explicit relationship information: Our next relationship detection mechanism relies on *explicit relationships* that directly communicate the programmer's view of his data structure to improve the precision. In a relational database, explicit relationships are defined in the form of foreign keys (FKs), which encode the precise relationship between two tables, based on primary keys (PKs). Interestingly, we can also extract a notion of foreign keys when relating DB rows to files: in some apps, the name of the file corresponds to the PK of the row to which it refers. Foreign keys encode the directionality of relationships, specifying for instance the difference between a "has-a" relationship and an "is-part-of" relationship. If node A has an FK to node B, then Pebbles adds the edge $A \rightarrow B$ (overriding any pre-existing bi-directional edge detected from data flow propagation). In this way, foreign keys are precise but limited in coverage because they require programmers to specify them explicitly.

Increasing recall: Pebbles relies on one final relationship detection mechanism, *access relationships*. Access relationships can be seen as similar to data relationships, but while data relationships identify relationships as they are written to storage, access relationships identify relationships as they are read. Consider the case where an application has some data in memory that has not been synced to stable storage (and therefore is not yet tainted with any node's GUID). The app uses the data to generate the index for key-value object A and also writes that data into database row B. In the absence of explicit relationship information, we would hope that data propagation would detect the relation; however, it cannot because there is no data flow relationship when the data is written. We call this situation a *parallel write*, and resolve it by detecting data flow relationships when data is read

Property 4.1. Apps define explicit relationships through FKs in DBs, XML hierarchies, or FS hierarchies

Property 4.2. The SQLite database is the hub of all persisted data storage and access

Object Graph Construction Algorithm:

1. Data propagation: If data from A is written to B, then $A \leftrightarrow B$
2. If possible, refine $A \leftrightarrow B$ to $A \rightarrow B$ using Prop 4.1
3. Access propagation: If data from A is used to read B, then $A \leftrightarrow B$
4. If possible, refine $A \leftrightarrow B$ to $A \rightarrow B$, again using Prop 4.1
5. Utilize Prop 4.2, eliminating access based data propagation relationships that do not include any DB nodes.

Fig. 5: **Object Graph Construction Rules.**

in from storage: if data tainted with node A's GUID is used to access (read) node B, Pebbles adds $A \leftrightarrow B$ to the object graph. Again, this process is agnostic to the storage abstraction that the data is stored in, and relies only on data flow within the app. Access relationships can become an even greater source of imprecision than data relationships. For example, one could use data from one row, such as a timestamp, to select all the rows with that timestamp. Does that imply that all those rows should be considered as one object? Probably not.

Graph Generation Algorithm: Fig. 5 defines the algorithm used to construct the object graph, based on the observation that the DB is the hub of all persisted data. Step (1) leverages data flow propagation to construct a base graph, while (2) refines that graph by applying explicit relationship information. Step (3) applies access based data flow propagation to increase recall, and (4) again refines that graph with explicit relationship information. §7.1 evaluates LDO construction accuracy and precision in detail.

4.3 LDO Construction and Semantics

After constructing the object graph using the above semantics, Pebbles extracts the LDOs. Within the graph, an LDO is defined as the set of reachable nodes starting with a given node (the root of the object). Consider the email graph (Fig. 4), one can define a number of LDOs: an `Account` LDO, rooted in one `Account`-table row and containing multiple instances of five other row types, two files, and one XML entry; an `Email` LDO, rooted in one `Message`-table row and containing another row and one file, and so on. Although one LDO of each type is defined in the figure, in reality, there would be as many LDOs as there are instances of that type.

It is possible and correct for a single node to be part of multiple otherwise separate LDOs, in which case we say that the LDOs *overlap*. Consider, for instance, stateful accumulators (e.g. counts or sums over objects, stored in other objects), common resources (e.g. cache files that contain information about multiple objects), or log files.

Interface	Returned Objects
`getLDOContent(GUID, relevantOnly)`	LDO rooted at `GUID`
`getParentLDOs(GUID, relevantOnly)`	LDOs that contain `GUID`

Table 1: **The Pebbles API for Accessing LDOs.**

Pebbles exposes LDOs to protection tools via the Pebbles API, which consists of two functions (Table 1). `getLDOContent` returns the LDO rooted at the given GUID and `getParentLDOs` returns the LDOs containing the given GUID. Protection tools may specify with each call if only LDOs that may be relevant to the end-user should be returned.

4.4 From User-Level Objects to LDOs

Both of these API methods require an "object of interest" as a parameter. Pebbles provides a framework for protection tools to allow users to directly select an object of interest (from the user interface), and then use that object for future API calls. In this approach, a user enables a "marking mode" from a device-wide menu item, and then touches the item that they are interested in. Through taint tracking, we can determine the internal GUID for the object that was selected, and return that GUID back to the protection tool. This feature makes designing user-centric protection tools very easy: the tool need not concern itself with determining which objects to protect.

The mechanisms described thus far are useful for building a graph of all of an application's objects, but does not yet include a way to identify those objects that are relevant to users. For instance, in our email application there is another table, "sync_state," that stores how recently an account was synchronized with the server. Sync_state should clearly not be considered its own LDO, as its existence is essentially hidden from the end-user – the user will likely consider whatever data is stored here as, logically, part of the account. Pebbles leverages its system-wide taint tracking to identify which nodes in the object graph are directly displayed on the screen, Pebbles marks those objects (and other LDOs of the same type) as *relevant*. If an object is not relevant, then Pebbles will not allow it to be the root node of an LDO, instead including it as a member of the nearest parent node displayed on the screen.

5 Pebbles-based Tools

To showcase the value of Pebbles, we built four different applications that leverage its object graph.

5.1 Breadcrumbs: Auditing Object Deletion

Motivated by Scenario 1 in §2.1, Breadcrumbs lets users audit the deletion of their objects – such as emails

Algorithm 1 Breadcrumbs Pseudocode

function WASFULLYDELETED(LDO l) $B \rightarrow$
  for all getLDOContent(l) as x **do**
    if x exists still **then** Add $x \rightarrow B$
    end if
  end for
  for all B as x **do**
    Display x and getParentLDOs(x) to the user
  end for
end function

or documents – by their applications. It uses Pebbles's primitives to track objects as they are being deleted and identify any breadcrumbs left behind by the application.

Users mark objects to audit for deletion (using Pebbles's object marking functionality), and then delete the object through their unmodified applications. They then open the Breadcrumbs application, which shows any persisted data related to recently tracked objects. In this way, users are not inundated with notifications about deletions and instead are only being presented with auditing information upon request. Fig. 6 shows a screenshot of Breadcrumbs's output when the user deletes an email in the Android email application. It shows the attachment file left behind and provides meaningful information about the leakage. A brief predefined interval after the user deletes a tracked object, Breadcrumbs destroys all relevant auditing information to protect the confidentiality of the partially deleted object.

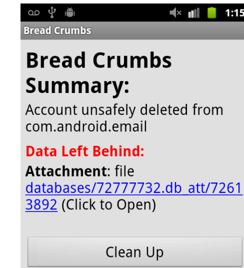


Fig. 6: **Breadcrumbs.**

Algorithm 1 shows how Breadcrumbs uses Pebbles's APIs to obtain all information necessary to identify and provide meaningful information about data left behind. Given a selected UI object, Pebbles identifies the GUID of the LDO represented by that LDO (as described in the previous section), and then Breadcrumbs calls `getLDOContent` to get all of its parts. For any part that still exists in persistent storage – the attachment file in this case – it displays meaningful metadata about that node. For example, instead of just showing the file's path, which can be nondescript, Breadcrumbs uses Pebbles's `getParentLDOs` function to retrieve the parent node, presumably a row. It displays the row's table name ("*Attachment*" in Fig.6), providing more context for information left behind. While the specific user interface we chose for Breadcrumbs can be improved, this example underscores the great value protection tools like Breadcrumbs can draw from understanding application-level object structures.

Our evaluation of Breadcrumbs on 50 apps (§7.3), reveals that incomplete deletions are surprisingly common:

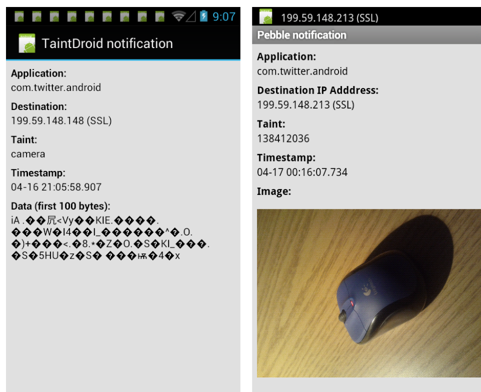


Fig. 7: **Alert Screenshots.** (L): TaintDroid, (R): PebbleNotify.

18/50 apps leave breadcrumbs or refuse to delete objects from the local device.

Breadcrumbs could also be a useful tool for developers. A developer could proactively use Breadcrumbs to ensure that they are responsibly handling their user's data.

5.2 PebbleNotify: Tracking Object Exfiltration

Inspired by TaintDroid's data exfiltration tool [11], we built PebbleNotify, a tool that tracks exfiltration at a more meaningful object level. TaintDroid reveals data exfiltration at a coarse granularity: it can only tell a user that *some* data from *some* provider was exfiltrated from the device, but not the specific data that was leaked. For instance, consider a cloud-based photo editing application. A user might expect this application to upload the photo being edited to a server for processing; however, he may be interested in checking that no other photos are exfiltrated. Shown in the left hand side of Fig.7, TaintDroid would warn the user that data related to *some* photo was uploaded, but not *which* photo or *how many* photos. PebbleNotify is a 500 line of code application built atop Pebbles that interposes on the same taint sinks as TaintDroid, but provides object-level warnings. §6 describes in somewhat greater detail the modifications that we made to TaintDroid to track individual objects with high precision. Shown in the right hand side of Fig.7, it leverages application-level data structures exposed by Pebbles to give users meaningful, fine-grained information about their leaked objects.

5.3 PebbleDIFC: Object Level Access Control

As a logical extension to PebbleNotify, consider the case where rather than monitor the exfiltration of sensitive data, users want to prevent specific apps from having access to it. For example, in our previous example of a user using a cloud-based photo editing application, perhaps the user would rather simply prevent that photo editing app from having any access whatsoever to sensitive photos. PebbleDIFC supports this use-case by interposing on Android content providers, the mechanism used to share data between apps.

PebbleDIFC allows users to select individual objects that are sensitive, and then prevent them from being shared with other applications (in this case, photos). As with the rest of our protection tools, PebbleDIFC's implementation is straightforward. Before returning an object from a content provider, PebbleDIFC checks a table that maps apps to hidden objects, and prevents access to hidden objects.

5.4 *HideIt*: Object Level Hiding

Whereas PebbleDIFC allows objects to be permanently hidden from specific apps, *HideIt* supports a slightly different use case: allowing objects to be selectively hidden from all apps on the device, and then re-displayed at some later point, and perhaps hidden again later on. When objects are hidden (again, using Pebbles's marking mode), they are encrypted, and any record of their existence is filtered, by interposing on storage APIs. When objects are un-hidden, they are decrypted, and no longer filtered from API results. *HideIt* is intended for use-cases where small amounts of data need to be infrequently hidden from prying eyes, for instance, a parent lending their phone to their child.

5.5 Other Pebbles-based Tools

Although we designed and implemented Pebbles for Android, we believe that its object recognition mechanisms are applicable to other environments where a database is used as the hub of storage. In particular, we can imagine applying Pebbles as a software engineering tool to help developers understand either current or legacy applications where the database is the storage hub. A developer could use Pebbles to explore undocumented systems that do not make use of modern abstractions such as object relational mappers that would make the system easy to understand or to determine whether an application conforms to best practices and alert the developer if not. Understanding data structure from below the application could also enable testing tools and policy compliance auditing tools for cloud services [36]. We leave investigation of such applications for future work.

6 Implementation

We implemented Pebbles and each of the four above protection tools on Android 2.3.4 and TaintDroid 2.3.4. For Pebbles, we modify the SQLite, XML key/value store (a.k.a. SharedPreferences), and Java file system API to extract explicit structure, to intercept read/write/delete operations, and to register relationships. We also make several key changes to the TaintDroid tracking system, which we release as open source (`https://systems.cs.columbia.edu/projects/os-abstractions`). We next review our TaintDroid changes, after which we describe some implementation-level details of object graph creation.

TaintDroid Changes. To support Pebbles, we made three modifications to TaintDroid: (1) we increase the number of supported taints from 32 to several million, (2) we implement multi-tainting to allow objects to have an arbitrary number of taints simultaneously, and (3) we implement fine-grained tainting. The first two TaintDroid changes are necessary to track every row, file, and XML element with a separate taint and are implemented with a technique recently proposed in the context of another taint tracking system [26]. We omit the details here for space reasons.

The third TaintDroid change is motivated by massive taint explosion that we observed due to TaintDroid's coarse-grained tracking. Specifically, TaintDroid stores a single taint tag per String and Array [11]. Deemed a performance benefit in the paper, this coarse-grained tracking is unusable in Pebbles: we observed extremely imprecise object recognition and application-wide LDOs due to this poor granularity. As one example, CWMoney, a personal finance application, has an internal array that holds selection arguments used in database queries. This causes all nodes selected by that query to be related, defeating any hopes of object precision.

To address this problem, we modify TaintDroid to add fine-grained tainting of individual Array and String elements. To implement fine-grained tainting we add a shadow buffer to the Dalvik ArrayObject that contains the taint of each element in the array. If implemented naively, the shadow arrays would likely double the memory required for each array. To minimize the memory overhead from the shadow arrays we allocate the shadow array only when a tainted element is inserted into the array. This same optimization is implemented in [8]. Intuitively, only a small fraction of arrays in an device's memory should contain tainted elements (3-5% according to our evaluation). §7.2 shows that this lazy shadow array allocation significantly reduces the memory overhead of precise fine-grained tainting. We release our changes open source as a patch for TaintDroid.

Object Graph Implementation. The Pebbles graph is populated incrementally during application execution and persisted in a central database on the data partition so the graph does not need to be regenerated on each reboot. Applications interact with the Pebbles API through the Pebbles Object Manager that runs as part of the central system server process. Graph edges are generated on read and write operations to SQLite, shared preferences, and the file system. On read and write operations that generate new edges, requests for edge registration are placed on a queue within the application's memory space. This lets Pebbles perform bulk asynchronous registrations off of the main application thread improving application interactivity even during periods of heavy edge creation. In its current implementation the registra-

tion queue is not persisted to stable storage so it will be lost on application crashes or restarts. This is a potential attack vector that does not fall under the threat model for non-malicious applications.

7 Evaluation

We evaluate Pebbles over 50 popular applications downloaded from Google's Android market on a Nexus S running our modified version of Android 2.3.4. We seek answers three key questions:

Q1 *How accurate and precise is object identification in Pebbles?*

Q2 *What performance overhead does it introduce?*

Q3 *How useful are Pebbles and the tools running atop?*

Application Workloads. We chose 50 test applications from the top free apps within 10 different Google Play Store categories, including Books and Reference, Finance, and Productivity. We looked at the top 30 most popular applications within each category (by number of installs) and selected those that used stable storage. We also added a few open-source applications (e.g., OINote). The resulting list included: Email (Android's default email app), OINote (open-source note app), Browser (Android's default), CWMoney (personal finance app), Bloomberg (stocks app), and PodcastAddict (podcast app). For each application, our workload involved exercising it in natural ways according to manual scripts. For example, in Wunderlist, a todo list app, we created multiple lists, added items to each list, and browsed through its functions.

7.1 Pebbles Precision and Recall (Q1)

We measure the precision and recall of our object recognition by identifying how closely LDOs match real, application-level objects as users perceive them. We manually identified 68 potentially interesting LDO types across 50 popular applications (e.g., individual emails, folders, and accounts in the default email app; individual expenses, expense categories, and accounts in the CWMoney financial app). We evaluated whether Pebbles correctly identifies those objects (no leakage or over-inclusions). Recall measures the percentage of LDOs recognized without leakage; precision measures the percentage of LDOs recognized without over-inclusion.

To establish ground truth about LDO structure, we first populated the application with data and took a snapshot of the phone's disk, S_1, prior to creating the target object. Then, we created the object and took a second snapshot of the disk, S_2. The ground truth is the diff between S_2 and S_1 after manually excluding differences that are unrelated to the objects (e.g., timestamps in log files that differ between the two executions). We then exercised the application as thoroughly as possible so as to capture any edges that Pebbles might detect. To measure accuracy, we compare Pebbles-recognized LDOs to the ground truth; if identical, we declare accurate recognition for that application and object.

		Pebbles		**File Tainting Only**	
Application	**LDO**	**Detected**	**Precise**	**Detected**	**Precise**
Email	Account	Y	Y	Y	**N**
	Mailbox	Y	Y	Y	**N**
	Email	Y	Y	Y	**N**
OINote	Note	Y	Y	Y	**N**
Browser	History Item	Y	Y	Y	**N**
	Bookmark	Y	Y	Y	**N**
CWMoney	Account	Y	Y	Y	**N**
	Category	Y	Y	Y	**N**
	Expense	Y	Y	Y	**N**
Bloomberg	Stock	**N**	Y	Y	**N**
	Chart	Y	Y	Y	**N**
Podcast	Podcast	Y	Y	Y	**N**
	Episode	**N**	Y	Y	**N**
50 Total	68 Total	62/68 (91%)	66/68 (97%)	68/68 (100%)	0/68 (0%)

Table 2: **LDO Precision and Recall.** Sample applications and objects tested for object recognition precision and recall. "Y" indicates that an LDO was identified without leakage (column "Detected") or without over inclusion (column "Precise"). If an LDO has "Y" in both columns, its recognition is deemed correct. As expected, Pebbles performs far better than a straw man approach of treating entire files as a single LDO.

Table 2 shows whether Pebbles correctly and precisely detects these LDOs. For comparison, we also evaluated the precision and recall of a basic approach, which represents perhaps the current state of the art: detecting relationships between files using just taint tracking and not using additional file structure to refine the granularity of objects. Pebbles correctly identifies 60 of the 68 objects across these 50 apps, without requiring any program modifications. Of the eight incorrectly identified objects, six were not correctly detected and two were not precise.

In each case that Pebbles failed to properly detect all components of the object (i.e., where it failed in recall), the leakage was due to a non-standard database specification. For instance, in the case of the app "ColorfulBudget", users can group expenses into categories, but Pebbles did not always properly detect the relationship between an expense and its category. Best practices would dictate that in such a case, all categories would be listed in a single table with a primary key (PK), and then each expense would contain a foreign key (FK) to reference the category's PK [4]. Traditionally this PK is an integer, to significantly increase lookup speed and decrease the amount of space needed to store any references to it [4]. However, in its current implementation, this app uses the actual name of the category as a key into the category table, without declaring such a dependency. Therefore, if a new category is created simultaneously with the creation of a new expense, we will experience a parallel write:

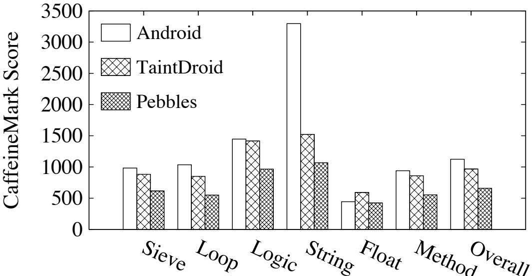


Fig. 8: **Java Microbenchmarks.** Overheads of the modified TaintDroid on the Java runtime with CaffeineMark, a standard Java benchmark. Higher values are better. Overheads on top of TaintDroid are 28-35%.

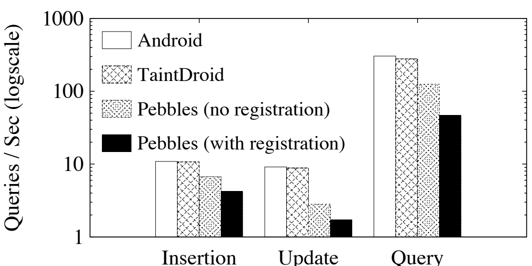


Fig. 9: **SQLite Microbenchmarks.** Overheads for various queries without and with relationship registrations.

there will be no data dependence when the category is inserted and when the expense is inserted, since the category did not yet exist in storage. Moreover, since the relationship is not declared in the app schema as an FK, explicit relationship mechanism will not detect it.

While our access-based technique will largely eliminate this problem, there is still a gap when data is written but never read back. In these scenarios, such relationships could never be detected. Had these apps explicitly declared their DB relationships (e.g., in the above case by referencing each category by its PK), Pebbles would accurately recognized the objects.

As an example of Pebbles failing in precision (i.e., including additional objects as part of an LDO), consider the "Evernote" note taking app. Each time a notebook is updated, text in a SharedPreferences node is updated to reflect the newest notebook, creating a data dependency between the SharedPreference and the notebook. In this way, each notebook can become related to each other because Pebbles currently does not break data dependencies when text is updated. The only way that relations are broken in Pebbles is if an explicit relationship exists and is removed.

Without requiring any modifications to applications, Pebbles is able to achieve up to 91% recall or 97% precision. The straw man approach of utilizing only taint tracking (without knowledge of file structure) showed perfect recall (100%), and a complete failure in precision (0%). In other words, there were *no cases* of a single logical object stored in a single file. Overall, our results confirm that an unsupervised approach to application-level object recognition from within the OS works well, especially if schemas are relatively well-defined.

7.2 Performance Evaluation (Q2)

To evaluate Pebbles performance overheads, we ran two types of benchmarks: (1) *microbenchmarks*, which let us stress various components of our system, such as the computation and SQLite plugins; and (2) *macrobenchmarks*, which let us quantify our system's performance impact on user-visible application latency. Pebbles is built atop the taint tracking system TaintDroid [11], with several modifications made to increase taint precision (as discussed in §6). Therefore, we evaluate the performance overhead of Pebbles in comparison to both TaintDroid and to a stock Android device.

Microbenchmarks. Our first experiments evaluate the overhead of Pebbles with the Java benchmark CaffeineMark 3.0 [27] and are shown in Fig. 8. We ran the six computational benchmarks and find that Pebbles decreases the score by 32% compared to TaintDroid, which itself decreases the score by 16% compared to Android. The majority of this overhead comes from modifications to support more than 32 taints in Pebbles: TaintDroid combines tags by bitwise OR'ing, but Pebbles supports 2^{32} distinct taint markings, which are maintained in a lookup table. Pebbles also stores taint tags per individual array element, whereas TaintDroid stores only one taint tag per array, creating an additional overhead for Pebbles array-heavy benchmarks.

Pebbles also incorporates modifications to SQLite to detect and register relationships between rows with the Pebbles service. To evaluate the overhead, we compared the latency of simple, constant-size `SELECT`, `INSERT`, and `UPDATE` queries on an Pebbles-enabled Android versus Android. Fig. 9 shows query overheads when the query involves a relationship registration (59-168%) and when it does not (158-553%). No-registration queries – the cheapest to Pebbles – will likely be the common case for read-mostly workloads. For example, a document may be read many times, but relationship registration occurs only once. Moreover, batching and asynchronous-registration optimizations will likely help alleviate the overheads. The XML-based key/value store exhibits similar behavior, although we suppress concrete results.

Application-Level Performance. The above workloads are micro-benchmarks that stress the various components but do not necessarily relate to user-perceived performance impacts. To measure the impact of Pebbles on user-perceived interactivity, we evaluated the runtimes for various operations with three popular applications: Email, Browser and OINote. For Email, we look at app launch times and email reads; for Browser, we load the simple IANA homepage and the rich CNN and Google News pages over a local network; and for OINote we

App	Activity	Base	TDroid	Pebbles	Overhead
Email	Launch	196.8	202.1	260.0	63.2 ±1.11
	Load Email	211.6	253.6	463.6	252.0 ±1.64
OINote	Launch	182.6	229.4	219.7	37.2 ±1.58
	Load Note	59.5	70.2	84.9	25.4 ±0.14
Browser	Launch	96.5	124.0	148.1	51.6 ±1.63
	Load (iana)	154.0	209.3	395.3	241.4 ±2.26
	Load (CNN)	778.9	862.7	1443.1	664.2 ±17.56
	Load (GNews)	951.3	1023.5	1311.2	359.9 ±10.75

Table 3: **Application Performance.** Operation runtimes and overheads in milliseconds. 95% confidence interval shown for overhead. Base is the Android baseline, TDroid is TaintDroid.

read a note. All network access occurred over USB tethering to a host running a caching proxy; timing information excludes cache warmup. Table 3 shows the results in milliseconds. In almost all of the cases, overhead was less than 250ms. We saw more overhead and variation when rendering multimedia heavy web pages.

Memory Overheads. The modifications to TaintDroid to add fine grained tainting adds a memory overhead to the running system. We measure system wide memory usage while exercising three applications (Email, OINote, and Browser) with a similar workload as above. Without lazy memory allocation of array taint vectors (see §6), Pebbles's system-wide memory overheads are high: 188MB, 70MB, and 119MB, respectively, compared to TaintDroid. With lazy memory allocation, Pebbles exhibits much lower system-wide overheads: 34MB, 16MB, and 29MB, respectively. Although still higher than TaintDroid's own overhead of around 7MB for these applications, we believe Pebbles overheads are acceptable given devices' increased memory trends.

7.3 Case Study Evaluation (Q3)

Breadcrumbs. Using our Breadcrumbs prototype we evaluated deletion practices of 68 types of LDOs across 50 applications. Of the 50 applications, 18 of them exhibited some type of deletion malpractice.

Table 4 shows sample deletion malpractice. There were several cases where data from one LDO was written into another another and not cleaned up later. There were also several applications that did not delete items at the users' request, instead simply removing them from the user interface. We observed this in applications that heavily rely on cloud storage such as Wunderlist, a popular cloud-backed todo list application.

PebbleNotify. To evaluate PebbleNotify, we compared its output to that of TaintDroid Notify. When TaintDroid Notify detects that data tainted with a value from one of the selected sources is exfiltrated, it notifies the user with the application that is responsible for the network connection, the destination, the data source, the timestamp, and the first 100 bytes of the packet. This is useful metadata but it won't help a user learn specific information about the data being exfiltrated such as which picture or specific contact is leaving the device. We found that PebbleNotify was more informative because it shows a summary of the data being exfiltrated, and not just the metadata presented by TaintDroid Notify. PebbleNotify was particularly useful in the case of image exfiltration because it displays a thumbnail of the image being sent.

Application	Object Deletion Leakage
Email	Attachments remain after email/account deletion
ExpenseManager	Expenses remain after associated category deleted
Evernote	Notes/notebooks remain in database after deletion
On Track	Measurements remain after deleting category
14 other apps	21 LDO types unsafely deleted

Table 4: **Breadcrumbs Findings.** Shows samples of unsafe deletion in various applications.

PebbleDIFC. We integrated PebbleDIFC with the Android Media Provider and evaluated it by using it to mark several photographs on our device as sensitive (i.e., to prevent them from being shared). We then verified that those photos were not visible to applications other than the default Gallery application. We found that for this use case, PebbleDIFC has perfect accuracy: every photo that was marked was hidden, and no additional photos were hidden.

HideIt. We evaluated *HideIt* against many applications and largely found it to be effective. In our evaluation, we interacted with the application, populated it with data, and then marked a subset of the data as private so the application no longer had access. Interestingly, in most cases apps behaved as hoped when individual data objects were hidden and then again returned. There were however several cases where apps crashed when they expected some data to still exist, but was removed. We are interested in performing further investigations of the applicability of *HideIt*.

7.4 Anecdotal User Experience

To gain experience with Pebbles, the primary author carried it on his Nexus S phone for about a week. He primarily used the Email, Browser, Gallery, Camera, and PodcastAddict apps. We report two anecdotal observations from this experience. First, applications exhibit noticeable overhead during periods of intense I/O, such as on initial launch or when applications populate or refresh local stores. During regular operation we observed overheads that are anecdotally similar to ones exhibited by running Android 4.1 (a 2012 OS) on our Nexus S (a 2010 device). Second, to check if object recognition remains accurate over time, we examined at the end of the week the structures of a sample of the objects in our applications (e.g., emails, folders, photos, browser histories, and podcasts). We saw no evidence that object recognition degraded over time due to taint explosions or other po-

tential sources of imprecision for Pebbles. Objects grew naturally; email folders grew in size to include relevant new email objects and they remained accurate.

7.5 Summary

Overall, our results show that: Pebbles is quite accurate in constructing LDOs in an unsupervised manner (*Q1*), performance remains reasonable when doing so (*Q2*), and data management tools can benefit from Pebbles to provide useful, consumer-grade functions to the users (*Q3*). In our experience, Pebbles either consistently identifies objects of a particular type (e.g., all emails, all documents, etc.), or it does not. Whether it works depends largely upon the application's own adherence to some common practices (described in the next section). When Pebbles works for all object types of an application, Pebbles can provide the desired guarantees under our threat model. And even when Pebbles is incomplete, it can still support transparency applications, improving visibility into data (mis)management of applications. Our accuracy results show that Pebbles discovers all object types in 42 out of 50 applications correctly (no over-inclusions/leakages). We leave development of tools to identify whether an application matches the Pebbles assumptions for future work.

8 Discussion

Pebbles leverages the structure inherently present in the storage abstractions commonly used on Android to identify LDOs. More formally, Pebbles assumes the usage of the following best practices:

R1: *Declare database schemas in full:* Given that the database is becoming the central point of all storage in modern OSes, having a well-defined database schema is important and natural. 42/50 apps we have evaluated in §7.1 meet such requirements sufficiently for Pebbles to work perfectly for them.

R2: *Use the database to index data within other storage systems:* A common programming pattern is to create a parent object (e.g., a message) in the database, obtain an auto-generated primary key, and then write any children objects (such as message body, attachment files) using the PK as a link. 47/50 apps use this pattern. We strongly recommend it to any programmers who need to store data outside the DB.

R3: *Use standard storage libraries or implement Pebbles storage API:* To avoid precision lapses, we recommend that apps use standard storage abstractions. As §3 shows, most apps already adhere to this practice: most apps use exclusively OS-embedded abstractions.

Relative to our evaluation of 50 apps, 39/50 adhere with all three recommendations, and 50/50 adhere with at least one of them. Pebbles' performance could suffer for apps that do not follow any of these recommendations. However, we believe that each recommendation is sufficiently intuitive and rooted in best practices to not impose undue burden.

9 Related Work

Taint Tracking for Protection and Auditing. Taint tracking systems (such as [3, 6, 17, 24, 31, 46, 49]) implement a dynamic data flow analysis that has been applied to many different context such as privacy auditing [6, 11, 48], malware analysis [24], and more [3, 49]. TaintDroid [11] provides taint tracking of unmodified Android applications through a modified Dalvik VM, a system that Pebbles builds upon for its object graph construction. To our knowledge, Pebbles is the first system to use taint tracking to discover data semantics of objects and provide a higher level abstraction with which to reason about and enforce such security properties.

Several systems utilize taint tracking to provide fine grained data protection and auditing. In each of these cases, however, a burden lies on the application developers to add hooks to identify relevant data structures to protection tool developers – a burden that could be lifted by Pebbles. For instance, CleanOS aims to minimize data exposure on a mobile device by automatically encrypting its "sensitive data objects" (SDOs) when not under active use [39]. The LDO abstraction is perhaps to some extent inspired by the SDO; however, SDOs must be manually specified by application developers, whereas LDOs are automatically identified and registered by Pebbles. Pebbles could be used to automatically identify SDOs, without requiring developer interaction.

Distributed information flow control (DIFC) systems such as Laminar [31], Asbestos [43], and Resin [46] let developers associate data with labels, and then allow either developers or end-users to specify security policies that apply to different labels. Taint tracking is performed during application execution to ensure that labels are propagated to derived data. Pebbles could be used to eliminate the need to statically annotate data with labels in code, instead automatically applying labels to LDOs as users request them. PebbleDIFC demonstrates the feasibility and power of such a system.

Related to taint tracking, data provenance [22, 23, 35] is close in spirit to logical data objects. It tracks the lineage of data (e.g., the user or process that created it). It has been proposed to identify the original authors of online information, to facilitate reproduction of scientific experiments [35], detect and avoid faulty data propagation in clouds [23], and others. It has to our knowledge never been used as an OS protection abstraction.

Fine-Grained Protection in Operating Systems. Many systems have been proposed in the past to support fine-

grained, flexible protection in operating systems. Some of the earliest OSes, such as Hydra [45] and Multics [32], provided immense protection flexibility to applications and users. Over time, OSes removed more and more flexibility, being considered too difficult for programmers. Our goal is to eliminate the programmer from the loop by having the OS identifying objects.

More recently, OS security extension systems, such as SELinux [34] and its Android version, SEAndroid [33], extend Linux's access control with flexible policies that determine which users and processes can access which resources, such as files, network interfaces, etc. Our work is complementary to these, being concerned with external attacks, such as thieves, shoulder surfing, or spying by a user with whom the device has been willfully shared. Our abstractions, might, however, apply to SEAndroid to replace its antiquated file abstraction.

Securing and Hiding Data. Many encryption systems exist, operating largely at one of two levels of abstraction: block level [1, 21, 42] and file level [14, 16]. A drawback to such encrypted file systems is that it forces users to consider data as individual files, while logically there may be multiple objects that the user is interested in in a single file. Pebbles allows protection tool developers to provide a far finer level of control (at the object level) than these existing systems (at the file level).

Some protection tools are already operating at a higher level of data abstraction. These applications, such as Vault-Hide [25] and KeepSafe Vault [19], allow users to hide specific types of data, including photos, contacts, and SMSes. However, they only plug into a handful of supported apps and cannot provide generic protection for all apps. Pebbles aims to effect a similar level of control, but without requiring specialized work by protection tool developers to support specific applications.

Inferring Structure in Semistructured Data. Discovering data relationships is a key aspect of our work. Other have worked on inferring data relationships in various context: foreign key relationships in databases to improve querying [30, 47] and file relationships in OSes to enhance file search [37]. However, Pebbles can also infer relations among files, as well as other higher-level storage abstractions within modern operating systems. To perform such broad relationship detection, Pebbles differs significantly from other relationship detection systems in that it also leverages taint tracking.

Cozzie et al. developed the Laika system [9] which uses Bayesian analysis to infer data structures from memory images. Pebbles differs from Laika in that it does not attempt to recover programmer defined data structures but to discover application-level data relationships from stable storage that would be recognizable and useful to an end user or developer.

10 Conclusions

We have described *logical data objects* (LDOs), a new fine-grained protection abstraction for persistent data designed specifically to enable the development of protection tools at a new granularity. We described our implementation of LDOs for Android with *Pebbles*, a system that automatically reverse engineers LDOs from application-level persisted data resources – such as emails, documents, or bank accounts. Pebbles leverages the structural semantics available in modern persistent storage systems, together with a number of mechanisms rooted in taint tracking, to construct and maintain an object graph that tracks these LDOs without introducing any new programming models or APIs.

We have evaluated Pebbles and four novel protection tools that use it, showing it to be accurate, and sufficiently efficient to be used in practice to identify and manage LDOs. We can envision many other useful applications of Pebbles, such as data scrubbing or malware analysis, and hope that LDOs will enable the development of these and other granular data protection systems.

11 Acknowledgements

We thank our shepherd, Landon Cox and the anonymous reviewers for their valuable feedback, and Emmett Witchel for his support and advice. This work was supported by DARPA Contract FA8650-11-C-7190; NSF grants CNS-1351089, CCF-1302269, CCF-1161079, CNS-0905246, and CNS-1228843; NIH U54 CA121852; R01 LM011028-01; and Google and Microsoft gifts.

References

[1] dm-crypt: Linux kernel device-mapper crypto target. `https://code.google.com/p/cryptsetup/wiki/DMCrypt`, 2013.

[2] Anand Basu. Facebook Apps Leak User Information. `http://www.reuters.com/article/2010/10/18/us-facebook-idUSTRE69H0QS20101018`, 2010.

[3] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[4] Michael Brackett. *Data Resource Design: Reality Beyond Illusion*. IT Pro. Technics Publications Llc, 2012.

[5] Monica Chew. Writing for the 98%, blog post. `http://monica-at-mozilla.blogspot.com/2013/02/writing-for-98.html`, 2013.

[6] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium (Sec)*, 2004.

[7] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium (Sec)*, 2005.

[8] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. Spandex: Secure password tracking for android. In *Proceedings of the USENIX Security Symposium (Sec)*, 2014.

[9] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[10] Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[11] William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[12] Roxana Geambasu, John P. John, Steven D. Gribble, Tadayoshi Kohno, and Henry M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[13] Google. Storage options — android developers. `http://developer.android.com/guide/topics/data/data-storage.html`.

[14] Valient Gough. encfs. `www.arg0.net/encfs`, 2010.

[15] GRSecurity. Homepage of pax. `http://pax.grsecurity.net/`.

[16] Michael Austin Halcrow. eCryptfs: An enterprise-class encrypted filesystem for linux. In *Proceedings of the Linux Symposium*, 2005.

[17] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2005.

[18] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2011.

[19] KeepSafe. Hide pictures - KeepSafe Vault. `https://play.google.com/store/apps/details?id=com.kii.safe`.

[20] Mary Madden and Aaron Smith. Reputation management and social media: How people monitor their identity and search for others online. `http://www.pewinternet.org/~/media/Files/Reports/2010/PIP_Reputation_Management_with_topline.pdf`, 2010.

[21] Microsoft Corporation. Windows 7 BitLocker executive overview. `http://technet.microsoft.com/en-us/library/dd548341(WS.10).aspx`, 2009.

[22] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2006.

[23] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[24] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[25] NQ Mobile Security. Vault-Hide SMS, Pics & Videos. `https://play.google.com/store/apps/details?id=com.netqin.ps`.

[26] Vasilis Pappas, Vasileios P. Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D. Keromytis. CloudFence: Data flow tracking as a

cloud service. In *Proceedings of the Symposium on Research in Attacks, Intrusions and Defenses*, 2013.

[27] Pendragon Software Corporation. Caffeinemark 3.0. http://www.benchmarkhq.ru/cm30/.

[28] Radia Perlman. File system design with assured delete. In *Proceedings of the IEEE International Security in Storage Workshop (SISW)*, 2005.

[29] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the USENIX Security Symposium (Sec)*, 2012.

[30] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A machine learning approach to foreign key discovery. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2009.

[31] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[32] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM (CACM)*, 1974.

[33] SEAndroid. SEforAndroid. http://selinuxproject.org/page/SEAndroid.

[34] SELinux. Selinux project wiki. http://selinuxproject.org/page/Main_Page.

[35] Margo Seltzer. Pass: Provenance-aware storage systems. http://www.eecs.harvard.edu/syrah/pass/.

[36] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. Bootstrapping privacy compliance in big data systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[37] Craig A.N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2005.

[38] Symantec Corporation. PGP whole disk encryption. http://www.symantec.com/whole-disk-encryption, 2012.

[39] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Mobile OS abstractions for managing sensitive data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[40] Yang Tang, Patrick P.C. Lee, John C.S. Lui, and Radia Perlman. FADE: Secure overlay cloud storage with file assured deletion. In *Proceedings of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2010.

[41] The Chaos Computing Club (CCC). CCC breaks Apple TouchID. http://www.ccc.de/en/updates/2013/ccc-breaks-apple-touchid, 2013.

[42] TrueCrypt Foundation. Truecrypt – free open-source on-the-fly encryption. http://www.truecrypt.org/, 2007.

[43] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems (TOCS)*, 2007.

[44] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, 2014.

[45] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM (CACM)*, 1974.

[46] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[47] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 2010.

[48] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings*

of the ACM Conference on Computer and Communications Security (CCS), 2013.

[49] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 2011.

Protecting Users by Confining JavaScript with COWL

Deian Stefan* Stanford | Edward Z. Yang Stanford | Petr Marchenko Google | Alejandro Russo† Chalmers | Dave Herman Mozilla

Brad Karp UCL | David Mazières Stanford

Abstract

Modern web applications are conglomerations of JavaScript written by multiple authors: application developers routinely incorporate code from third-party libraries, and *mashup* applications synthesize data and code hosted at different sites. In current browsers, a web application's developer and user must trust third-party code in libraries not to leak the user's sensitive information from within applications. Even worse, in the status quo, the only way to implement some mashups is for the user to give her login credentials for one site to the operator of another site. Fundamentally, today's browser security model trades privacy for flexibility because it lacks a sufficient mechanism for *confining untrusted code.* We present COWL, a robust JavaScript confinement system for modern web browsers. COWL introduces label-based mandatory access control to browsing contexts in a way that is fully backward-compatible with legacy web content. We use a series of case-study applications to motivate COWL's design and demonstrate how COWL allows both the inclusion of untrusted scripts in applications and the building of mashups that combine sensitive information from multiple mutually distrusting origins, all while protecting users' privacy. Measurements of two COWL implementations, one in Firefox and one in Chromium, demonstrate a virtually imperceptible increase in page-load latency.

1 Introduction

Web applications have proliferated because it is so easy for developers to reuse components of existing ones. Such reuse is ubiquitous. jQuery, a widely used JavaScript library, is included in and used by over 77% of the Quantcast top-10,000 web sites, and 59% of the Quantcast top-million web sites [3]. While component reuse in the venerable desktop software model typically involves libraries, the reusable components in web applications are not limited to just JavaScript library code—they further include network-accessible content and services.

The resulting model is one in which web developers cobble together multiple JavaScript libraries, web-based content, and web-based services written and operated by various parties (who in turn may integrate more of these resources) and build the required application-specific functionality atop them. Unfortunately, some of the many contributors to the tangle of JavaScript comprising an application may not have the user's best interest at heart. The wealth of sensitive data processed in today's web applications (*e.g.,* email, bank statements, health records, passwords, *etc.*) is an attractive target. Miscreants may stealthily craft malicious JavaScript that, when incorporated into an application by an unwitting developer, violates the user's privacy by leaking sensitive information.

Two goals for web applications emerge from the prior discussion: *flexibility* for the application developer (*i.e.,* enabling the building of applications with rich functionality, composable from potentially disparate pieces hosted by different sites); and *privacy* for the user (*i.e.,* to ensure that the user's sensitive data cannot be leaked from applications to unauthorized parties). These two goals are hardly new: Wang *et al.* articulated similar ones, and proposed new browser primitives to improve isolation within *mashups,* including discretionary access control (DAC) for inter-frame communication [41]. Indeed, today's browsers incorporate similar mechanisms in the guises of HTML5's iframe sandbox and postMessage API [47]. And the *Same-Origin Policy* (SOP, reviewed in Section 2.1) prevents JavaScript hosted by one principal from reading content hosted by another.

Unfortunately, in the status-quo web browser security architecture, one must often sacrifice privacy to achieve flexibility, and vice-versa. The central reason that flexibility and privacy are at odds in the status quo is that the mechanisms today's browsers rely on for providing privacy—the SOP, Content Security Policy (CSP) [42], and Cross-Origin Resource Sharing (CORS) [45]—are all forms of discretionary access control. DAC has the brittle character of either denying or granting untrusted code (*e.g.,* a library written by a third party) access to data. In the former case, the untrusted JavaScript might *need* the sensitive data to implement the desired application functionality—hence, denying access prioritizes privacy over flexibility. In the latter, DAC exercises no control over what the untrusted code does with the sensitive data—and thus prioritizes flexibility over privacy. DAC is an essential tool in the privacy arsenal, but *does not fit cases where one runs untrusted code on sensitive input,* which are the norm for web applications, given their multi-contributor nature.

In practice, web developers turn their backs on privacy in favor of flexibility because the browser doesn't offer

*Work partly conducted while at Mozilla.

†Work partly conducted while at Stanford.

primitives that let them opt for both. For example, a developer may want to include untrusted JavaScript from another origin in his application. All-or-nothing DAC leads the developer to include the untrusted library with a `script` tag, which effectively bypasses the SOP, interpolating untrusted code into the enclosing page and granting it unfettered access to the enclosing page's origin's content.[1] And when a developer of a mashup that integrates content from *other* origins finds that the SOP forbids his application from retrieving data from them, he designs his mashup to require that the user provide the mashup her login credentials for the sites at the two other origins [2]—the epitome of "functionality over privacy."

In this paper, we present COWL (Confinement with Origin Web Labels), a mandatory access control (MAC) system that confines untrusted JavaScript in web browsers. COWL allows untrusted code to compute over sensitive data and display results to the user, but prohibits the untrusted code from exfiltrating sensitive data (*e.g.,* by sending it to an untrusted remote origin). It thus allows web developers to opt for *both* flexibility and privacy.

We consider four motivating example web applications—a password strength-checker, an application that imports the (untrusted) jQuery library, an encrypted cloud-based document editor, and a third-party mashup, none of which can be implemented in a way that preserves the user's privacy in the status-quo web security architecture. These examples drive the design requirements for COWL, particularly MAC with *symmetric and hierarchical confinement* that supports *delegation.* Symmetric confinement allows *mutually* distrusting principals each to pass sensitive data to the other, and confine the other's use of the passed sensitive data. Hierarchical confinement allows any developer to confine code she does not trust, and confinement to be nested to arbitrary depths. And delegation allows a developer explicitly to confer the privileges of one execution context on a separate execution context. No prior browser security architecture offers this combination of properties.

We demonstrate COWL's applicability by implementing secure versions of the four motivating applications with it. Our contributions include:

- ► We characterize the shared needs of four case-study web applications (Section 2.2) for which today's browser security architecture cannot provide privacy.
- ► We describe the design of the COWL label-based MAC system for web browsers (Section 3), which meets the requirements of the four case-study web applications.
- ► We describe designs of the four case-study web applications atop COWL (Section 4).
- ► We describe implementations of COWL (Section 5) for the Firefox and Chromium open-source browsers; our evaluation (Section 6) illustrates that COWL incurs minimal performance overhead over the respective baseline browsers.

[1] Indeed, jQuery *requires* such access to the enclosing page's content!

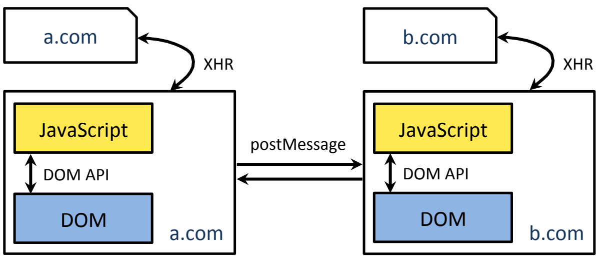


Figure 1: Simplified browser architecture.

2 Background, Examples, & Goals

A single top-level web page often incorporates multiple scripts written by different authors.[2] Ideally, the browser should protect the user's sensitive data from unauthorized disclosure, yet afford page developers the greatest possible flexibility to construct featureful applications that reuse functionality implemented in scripts provided by (potentially untrusted) third parties. To make concrete the diversity of potential trust relationships between scripts' authors and the many ways page developers structure amalgamations of scripts, we describe several example web applications, none of which can be implemented with strong privacy for the user in today's web browsers. These examples illustrate key requirements for the design of a flexible browser confinement mechanism. Before describing these examples, however, we offer a brief refresher on status-quo browser privacy polices.

2.1 Browser Privacy Policies

Browsing contexts Figure 1 depicts the basic building blocks of the current web security architecture. A *browsing context* (*e.g.,* a page or frame) encapsulates presentable content and a JavaScript execution environment (heap and code) that interacts with content through the *Document Object Model (DOM)* [47]. Browsing contexts may be nested (*e.g.,* by using iframes). They also may read and write persistent storage (*e.g.,* cookies), issue network requests (either implicitly in page content that references a URL retrieved over the network, or explicitly in JavaScript, using the `XMLHttpRequest` (XHR) constructor), and communicate with other contexts (IPC-style via `postMessage`, or, in certain cases, by sharing DOM objects). Some contexts such as Web Workers [44] run JavaScript but do not instantiate a DOM. We use the terms *context* and *compartment* interchangeably to refer to both browsing contexts and workers, except when the more precise meaning is relevant.

Origins and the Same-Origin Policy Since different authors may contribute components within a page, today's

[2] Throughout we use "web page" and "web application" interchangeably, and "JavaScript code" and "script" interchangeably.

status quo browsers impose a security policy on interactions among components. Policies are expressed in terms of *origins*. An origin is a source of authority encoded by the protocol (*e.g.,* `https`), domain name (*e.g.,* `fb.com`), and port (*e.g.,* `443`) of a resource URL. For brevity, we elide the protocol and port from URLs throughout.

The same-origin policy specifies that an origin's resources should be readable only by content from the same origin [7, 38, 52]. Browsers ensure that code executing in an `a.com` context can only inspect the DOM and cookies of another context if they share the same origin, *i.e.,* `a.com`. Similarly, such code can only inspect the response to a network request (performed with XHR) if the remote host's origin is `a.com`.

The SOP does not, however, prevent code from *disclosing* data to foreign origins. For example, code executing in an `a.com` context can trivially disclose data to `b.com` by using XHR to perform a network request; the SOP prevents the code from inspecting responses to such cross-origin XHR requests, but does not impose any restrictions on sending such requests. Similarly, code can exfiltrate data by encoding it in the path of a URL whose origin is `b.com`, and setting the `src` property of an `img` element to this URL.

Content Security Policy (CSP) Modern browsers allow the developer to protect a user's privacy by specifying a CSP that limits the communication of a page—*i.e.,* that disallows certain communication ordinarily permitted by the SOP. Developers may set individual CSP directives to restrict the origins to which a context may issue requests of specific types (for images or scripts, XHR destinations, *etc.*) [42]. However, CSP policies suffer from two limitations. They are *static:* they cannot change during a page's lifetime (*e.g.,* a page may not drop the privilege to communicate with untrusted origins before reading potentially sensitive data). And they are *inaccessible:* JavaScript code cannot inspect the CSP of its enclosing context or some other context, *e.g.,* when determining whether to share sensitive data with that other context.

postMessage and Cross-Origin Resource Sharing (CORS) As illustrated in Figure 1, the HTML5 `postMessage` API [43] enables cross-origin communication in IPC-like fashion within the browser. To prevent unintended leaks [8], a sender always specifies the origin of the intended recipient; only a context with that origin may read the message.

CORS [45] goes a step further and allows controlled cross-origin communication between a browsing context of one origin and a remote server with a different origin. Under CORS, a server may include a header on returned content that explicitly whitelists other origin(s) allowed to read the response.

Note that both `postMessage`'s target origin and CORS are purely discretionary in nature: they allow static selection of which cross-origin communication is allowed and which denied, but enforce no confinement on a receiving compartment of differing origin. Thus, in the status-quo web security architecture, a privacy-conscious developer should only send sensitive data to a compartment of differing origin if she completely trusts that origin.

2.2 Motivating Examples

Having reviewed the building blocks of security policies in status-quo web browsers, we now turn to examples of web applications for which strong privacy is not achievable today. These examples illuminate key design requirements for the COWL confinement system.

Password Strength Checker Given users' propensity for choosing poor (*i.e.,* easily guessable) passwords, many web sites today incorporate functionality to check the strength of a password selected by a user and offer the user feedback (*e.g.,* "too weak; choose another," "strong," *etc.*). Suppose a developer at Facebook (origin `fb.com`) wishes to re-use password-checking functionality provided in a JavaScript library by a third party, say, from origin `sketchy.ru`. If the developer at `fb.com` simply includes the third party's code in a `script` tag referencing a resource at `sketchy.ru`, then the referenced script will have unfettered access to both the user's password (provided by the Facebook page, which the library *must* see to do its job) and to write to the network via XHR. This simple state of affairs is emblematic of the ease with which naïve web developers can introduce leaks of sensitive data in applications.

A more skilled web developer could today host the checker script on her *own* server and have that server specify a CSP policy for the page. Unfortunately, a CSP policy that disallows scripts within the page from initiating XHRs to any other origins is *too inflexible,* in that it precludes useful operations by the checker script, *e.g.,* retrieving an updated set of regular expressions describing weak passwords from a remote server (essentially, "updating" the checker's functionality). Doing so requires communicating with a remote origin. Yet a CSP policy that permits such communication, even with the top-level page's same origin, is *too permissive:* a malicious script could potentially carry out a *self-exfiltration attack* and write the password to a public part of the trusted server [11, 50].

This trade-off between flexibility and privacy, while inherent to CSP, need not be fundamental to the web model. The key insight is that it is entirely safe and useful for an untrusted script to communicate with remote origins *before* it reads sensitive data. We note, then, the requirement of a confinement mechanism that allows code in a compartment to communicate with the network *until it has been exposed to sensitive data.* MAC-based confinement meets this requirement.

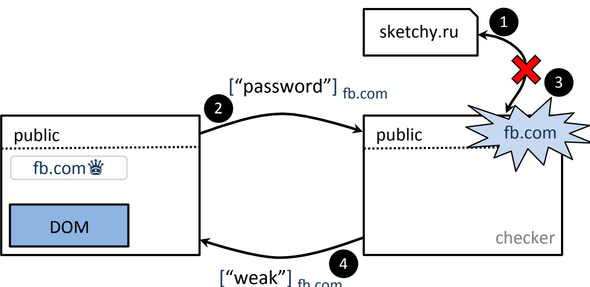


Figure 2: Third-party password checker architecture under COWL.

Figure 2 shows how such a design might look. In this and subsequent examples, rectangular frames denote compartments, arrows denote communication (either between a compartment and the network, or IPC-style between compartments), and events during execution are numbered sequentially in time. As we have proposed previously [49], compartments may be *labeled* (Section 3.1) with the origins to whose sensitive data they have been exposed. A compartment that has not yet observed sensitive data is denoted `public`; however, when it wishes to incorporate sensitive data, the compartment *raises* its label (at the cost of being more restricted in where it can write). We illustrate the raising of a label with a "flash" connoting the sensitivity of data being integrated. A compartment's *privilege* (Section 3.3), which specifies the origins for which a script executing in that compartment is trusted, is indicated by a crown. Here, a top-level page at `fb.com` encapsulates a password-checker script from a third-party origin in a new compartment. The label of the new compartment is initially `public`. First, in step (1), the checker script is free to download updated regular expressions from an arbitrary remote origin. In step (2), the top-level page sends the user's password to the checker script's worker using `postMessage`; the password is *labeled* `fb.com` to indicate that the data is sensitive to this origin (Section 3.2). In step (3) the checker raises its label to reflect that the context is about to be exposed to sensitive data from `fb.com` and inspects the password. When the label is raised, COWL atomically denies the context further access to the network in step (3).[3] However, the checker script is free to compute the result, which it then returns via `postMessage` to the top-level page in step (4); the result carries the label `fb.com` to reflect that the sender may be sending data derived from sensitive data owned by `fb.com`. Since the top-level page has the `fb.com` privilege, it can simply read the data (without raising its label).

[3] For clarity, we use `fb.com` as the label on the data. This label still allows the checker to send XHR requests to `fb.com`; to ensure that the checker cannot communicate with *any* origin, COWL provides fresh origins (see Section 3.3).

Encrypted Document Editor Today's web applications, such as in-browser document editors backed by cloud-based storage (*e.g.,* Google Docs), typically require the user to trust the app developer/cloud-based storage provider (often the same principal under the SOP) with the data in her documents. That is, the provider's server observes the user's data in cleartext. Suppose an organization wished to use an in-browser document editor but did *not* want to reveal its users' document data to the editor provider's server. How might the provider offer a privacy-preserving editor app that would satisfy the needs of such a privacy-conscious organization? One promising approach might be for the "customer" privacy-sensitive organization to implement a trusted document encryption service hosted at its own origin, distinct from that which hosts the editor app. The editor app could allow the user to specify a JavaScript "plugin" library she trusts to perform cryptography correctly. In this design, one origin serves the JavaScript code for the editor app (say, `gdocs.com`) and a different origin serves the JavaScript code for the cryptography library (say, `eff.org`). Note that these two origins may be *mutually distrusting.* `gdocs.com`'s script must pass the document's cleartext to a script from `eff.org` for encryption, but would like to confine the execution of the encryption script so that it cannot exfiltrate the document to any origin *other* than `gdocs.com`. Similarly, `eff.org`'s cryptography library may not trust `gdocs.com` with the cleartext document—it would like to confine `gdocs.com`'s editor to prevent exfiltration of the cleartext document to `gdocs.com` (or to any other origin). This simple use case highlights the need for *symmetric confinement:* when two mutually distrusting scripts from different origins communicate, *each must be able to confine the other's further use of data it provides.*

Third-Party Mashup Some of the most useful web applications are *mashups*; these applications integrate and compute over data hosted by multiple origins. For example, consider an application that reconciles a user's Amazon purchases (the data for which are hosted by `amazon.com`) against a user's bank statement (the data for which are hosted by `chase.com`). The user may well deem both these categories of data sensitive and will furthermore not want data from Amazon to be exposed to her bank or vice-versa, nor to any other remote party. Today, if one of the two providers implements the mashup, its application code must bypass the SOP to allow sharing of data across origin boundaries, *e.g.,* by communicating between iframes with `postMessage` or setting a permissive CORS policy. This approach forfeits privacy: one origin sends sensitive data to the other, after which the receiving origin may exfiltrate that sensitive data at will. Alternatively, a third-party developer may wish to implement and offer this mashup application. Users of such a *third-party mashup* give up their privacy, usually by simply

handing off credentials, as again today's browser enforces no policy that confines the sensitive data the mashup's code observes within the browser. To enable third-party mashups that do not sacrifice the user's privacy, we note again the need for an untrusted script to be able to issue requests to multiple remote origins (*e.g.,* `amazon.com` and `chase.com`), but to lose the privilege to communicate over the network once it has read the responses from those origins. Here, too, MAC-based confinement addresses the shortcomings of DAC.

Untrusted Third-Party Library Web application developers today make extensive use of third-party libraries like jQuery. Simply importing a library into a page provides no isolation whatsoever between the untrusted third-party code and any sensitive data within the page. Developers of applications that process sensitive data want the convenience of reusing popular libraries. But such reuse risks exfiltration of sensitive data by these untrusted libraries. Note that because jQuery requires access to the content of the entire page that uses it, we cannot isolate jQuery in a separate compartment from the parent's, as we did for the password-checker example. Instead, we observe that jQuery demands a design that is a mirror image of that for confining the password checker: we place the *trusted* code for a page in a separate compartment and deem the rest of the page (including the untrusted jQuery code) as untrusted. The trusted code can then communicate with remote origins and inject sensitive data into the untrusted page, but the untrusted page (including jQuery) cannot communicate with remote origins (and thus cannot exfiltrate sensitive data within the untrusted page). This refactoring highlights the need for a confinement system that supports *delegation* and *dropping privilege*: a page should be able to create a compartment, confer its privileges to communicate with remote origins on that compartment, and then give these privileges up.

We note further that any *library* author may wish to reuse functionality from another untrusted library. Accordingly, to allow the broadest reuse of code, the browser should support *hierarchical confinement*—the primitives for confining untrusted code should allow not only a single level of confinement (one trusted context confining one untrusted context), but arbitrarily many levels of confinement (one trusted context confining an untrusted one, that in turn confines a further untrusted one, *etc.*).

2.3 Design Goals

We have briefly introduced four motivating web applications that achieve rich functionality by combining code from one or more untrusted parties. The privacy challenges that arise in such applications are unfortunately unaddressed by status-quo browser security policies, such as the SOP. These applications clearly illustrate the need for robust yet flexible confinement for untrusted code in browsers. To summarize, these applications would appear to be well served by a system that:

- Applies mandatory access control (MAC);
- Is *symmetric, i.e.,* it permits two principals to *mutually* distrust one another, and each prevent the other from exfiltrating its data;
- Is *hierarchical, i.e.,* it permits principal A to confine code from principal B that processes A's data, while principal B can independently confine code from principal C that processes B's data, *etc.*
- Supports *delegation* and *dropping privilege, i.e.,* it permits a script running in a compartment with the privilege to communicate with some set of origins to confer those privileges on another compartment, then relinquish those privileges itself.

In the next section, we describe COWL, a new confinement system that satisfies these design goals.

3 The COWL Confinement System

The COWL confinement system extends the browser security model while leaving the browser fully compatible with today's "legacy" web applications.[4] Under COWL, the browser treats a page exactly like a legacy browser does unless the page executes a COWL API operation, at which point the browser records that page as running in *confinement mode,* and all further operations by that page are subject to confinement by COWL. COWL augments today's web browser with three primitives, all of which appear in the simple password-checker application example in Figure 2.

Labeled browsing contexts enforce MAC-based confinement of JavaScript at the granularity of a context (*e.g.,* a worker or iframe). The rectangular frames in Figure 2 are labeled contexts. As contexts may be nested, labeled browsing contexts allow hierarchical confinement, whose importance for supporting nesting of untrusted libraries we discussed in Section 2.2.

When one browsing context sends sensitive information to another, a sending context can use *labeled communication* to confine the potentially untrusted code receiving the information. This enables symmetric confinement, whose importance in building applications that compose mutually distrusting scripts we articulated in Section 2.2. In Figure 2, the arrows between compartments indicate labeled communication, where a subscript on the communicated data denotes the data's label.

COWL may grant a labeled browsing context one or more *privileges,* each with respect to an origin, and each of which reflects trust that the scripts executing within

[4] In prior work, we described how confinement can subsume today's browser security primitives, and advocated replacing them entirely with a clean-slate, confinement-based model [49]. In this paper, we instead prioritize incremental deployability, which requires coexistence alongside the status quo model.

that context will not violate the secrecy and integrity of that origin's data, *e.g.,* because the browser retrieved them from that origin. A privilege authorizes scripts within a context to execute certain operations, such as *declassification* and *delegation*, whose abuse would permit the release of sensitive information to unauthorized parties. In COWL, we express privilege in terms of origins. The crown icon in the left compartment in Figure 2 denotes that this compartment may execute privileged operations on data labeled with the origin `fb.com`—more succinctly, that the compartment holds the privilege for `fb.com`. The compartment uses that privilege to remain unconfined by declassifying the checker response labeled `fb.com`.

We now describe these three constructs in greater detail.

3.1 Labeled Browsing Contexts

A COWL application consists of multiple labeled contexts. Labeled contexts extend today's browser contexts, used to isolate iframes, pages, *etc.*, with MAC *labels*. A context's label specifies the security policy for all data within the context, which COWL enforces by restricting the flow of information to and from other contexts and servers.

As we have proposed previously [33, 49], a label is a pair of boolean formulas over origins: a *secrecy* formula specifying which origins may read a context's data, and an *integrity* formula specifying which origins may write it. For example, only Amazon or Chase may read data labeled ⟨`amazon.com` ∨ `chase.com`, `amazon.com`⟩, and only Amazon may modify it.[5] Amazon could assign this label to its order history page to allow a Chase-hosted mashup to read the user's purchases. On the other hand, after a third-party mashup hosted by `mint.com` (as described in Section 2.2) reads *both* the user's Chase bank statement data *and* Amazon purchase data, the label on data produced by the third-party mashup will be ⟨`amazon.com` ∧ `chase.com`, `mint.com`⟩. This secrecy label component specifies that the data may be sensitive to both parties, and without both their consent (see Section 3.3), it should only be read by the user; the integrity label component, on the other hand, permits only code hosted by Mint to modify the resulting data.

COWL enforces label policies in a MAC fashion by only allowing a context to communicate with other contexts or servers whose labels are at least as restricting. (A server's "label" is simply its origin.) Intuitively, when a context wishes to send a message, the target must not allow additional origins to read the data (preserving secrecy). Dually, the source context must not be writable by origins not otherwise trusted by the target. That is, the source must be at least as trustworthy as the target. We say that such a target label "subsumes" the source label. For example, a context labeled ⟨`amazon.com`, `mint.com`⟩ can send messages to one labeled ⟨`amazon.com` ∧ `chase.com`, `mint.com`⟩, since the latter is trusted to preserve the privacy of `amazon.com` (and `chase.com`). However, communication in the reverse direction is not possible since it may violate the privacy of `chase.com`. In the rest of this paper, we limit our discussion to secrecy and only comment on integrity where relevant; we refer the interested reader to [33] for a full description of the label model.

A context can freely *raise* its label, *i.e.,* change its label to any label that is more restricting, in order to receive a message from an otherwise prohibited context. Of course, in raising its label to read more sensitive data from another context, the context also becomes more restricted in where it can write. For example, a Mint context labeled ⟨`amazon.com`⟩ can raise its label to ⟨`amazon.com` ∧ `chase.com`⟩ to read bank statements, but only at the cost of giving up its ability to communicate with Amazon (or, for that matter, any other) servers. When creating a new context, code can impose an upper bound on the context's label to ensure that untrusted code cannot raise its label and read data above this *clearance*. This notion of clearance is well established [14, 17, 34, 35, 51]; we discuss its relevance to covert channels in Section 7.

As noted, COWL allows a labeled context to create additional labeled contexts, much as today's browsing contexts can create sub-compartments in the form of iframes, workers, *etc.* This functionality is crucial for compartmentalizing a system hierarchically, where the developer places code of different degrees of trustworthiness in separate contexts. For example, in the password checker example in Section 2.2, we create a child context in which we execute the untrusted checker script. Importantly, however, code should not be able to leak information by laundering data through a newly created context. Hence, a newly created context implicitly inherits the current label of its parent. Alternatively, when creating a child, the parent may specify an initial current label for the child that is *more* restrictive than the parent's, to confine the child further. Top-level contexts (*i.e.,* pages) are assigned a default label of `public`, to ensure compatibility with pages written for the legacy SOP. Such browsing contexts can be restricted by setting a `COWL-label` HTTP response header, which dictates the minimal document label the browser must enforce on the associated content.

COWL applications can create two types of context. First, an application can create standard (but labeled) contexts in the form of pages, iframes, workers, *etc.* Indeed, it may do so because a COWL application is merely a regular web application that additionally uses the COWL API. It thus is confined by MAC, in addition to today's web security policies. Note that to enforce MAC, COWL must mediate all pre-existing communication channels—even

[5] ∨ and ∧ denote disjunction and conjunction. A comma separates the secrecy and integrity formulas.

subtle and implicit channels, such as content loading—according to contexts' labels. We describe how COWL does so in Section 5.

Second, a COWL application can create labeled contexts in the form of *lightweight labeled workers (LWorkers)*. Like normal workers [44], the API exposed to LWorkers is minimal; it consists only of constructs for communicating with the parent, the XHR constructor, and the COWL API. Unlike normal workers, which execute in separate threads, an LWorker executes in the same thread as its parent, sharing its event loop. This sharing has the added benefit of allowing the parent to give the child (labeled) access to its DOM, any access to which is treated as both a read and a write, *i.e.,* bidirectional communication. Our third-party library example uses such a *DOM worker* to isolate the trusted application code, which requires access to the DOM, from the untrusted jQuery library. In general, LWorkers—especially when given DOM access—simplify the isolation and confinement of scripts (*e.g.,* the password strength checker) that would otherwise run in a shared context, as when loaded with `script` tags.

3.2 Labeled Communication

Since COWL enforces a label check whenever a context sends a message, the design described thus far is already symmetric: a source context can confine a target context by raising its label (or a child context's label) and thereafter send the desired message. To read this message, the target context must confine itself by raising its label accordingly. These semantics can make interactions between contexts cumbersome, however. For example, a sending context may wish to communicate with multiple contexts, and need to confine those target contexts with different labels, or even confine the same target context with different labels for different messages. And a receiving context may need unfettered communication with one or more origins for a time before confining itself by raising its label to receive a message. In the password-checker example application, the untrusted checker script at the right of Figure 2 exhibits exactly this latter behavior: it needs to communicate with untrusted remote origin `sketchy.ru` before reading the password labeled `fb.com`.

Labeled Blob Messages (Intra-Browser) To simplify communication with confinement, we introduce the *labeled Blob,* which binds together the payload of an individual inter-context message with the label protecting it. The payload takes the form of a serialized immutable object of type Blob [47]. Encapsulating the label with the message avoids the cumbersome label raises heretofore necessary in both sending and receiving contexts before a message may even be sent or received. Instead, COWL allows the developer sending a message from a context to specify the label to be attached to a labeled Blob; any label as or more restrictive than the sending context's current label may be specified (modulo its clearance). While the receiving context may receive a labeled Blob with no immediate effect on the origins with which it can communicate, it may only inspect the label, not the payload.[6] Only after raising its label as needed may the receiving context read the payload.

Labeled Blobs simplify building applications that incorporate distrust among contexts. Not only can a sender impose confinement on a receiver simply by labeling a message; a receiver can delay inspecting a sensitive message until it has completed communication with untrusted origins (as does the checker script in Figure 2). They also ease the implementation of integrity in applications, as they allow a context that is not trusted to modify content in some other context to serve as a passive conduit for a message from a third context that *is* so trusted.

Labeled XHR Messages (Browser–Server) Thus far we have focused on confinement as it arises when two browser contexts communicate. Confinement is of use in browser-server communication, too. As noted in Section 3.1, COWL only allows a context to communicate with a server (whether with XHR, retrieving an image, or otherwise) when the server's origin subsumes the context's label. Upon receiving a request, a COWL-aware web server may also wish to know the current label of the context that initiated it. For this reason, COWL attaches the current label to every request the browser sends to a server.[7] As also noted in Section 3.1, a COWL-aware web server may elect to label a response it sends the client by including a `COWL-label` header on it. In such cases, the COWL-aware browser will only allow the receiving context to read the XHR response if its current label subsumes that on the response.

Here, again, a context that receives labeled data—in this case from a server—may wish to defer raising its label until it has completed communication with other remote origins. To give a context this freedom, COWL supports *labeled XHR* communication. When a script invokes COWL's labeled XHR constructor, COWL delivers the response to the initiating script as a labeled Blob. Just as with labeled Blob intra-browser IPC, the script is then free to delay raising its label to read the payload of the response—and delay being confined—until after it has completed its other remote communication. For example, in the third-party mashup example, Mint only confines itself once it has received all necessary (labeled) responses from both Amazon and Chase. At this point it processes the data and displays results to the user, but it can no longer send requests since doing so may leak

[6] The label itself cannot leak information—COWL still ensures that the target context's label is at least as restricting as that of the source.

[7] COWL also attaches the current privilege; see Section 3.3.

information.[8]

3.3 Privileges

While confinement handily enforces secrecy, there are occasions when an application must eschew confinement in order to achieve its goals, and yet can uphold secrecy while doing so. For example, a context may be confined with respect to some origin (say, `a.com`) as a result of having received data from that origin, but may need to send an encrypted version of that data to a third-party origin. Doing so does not disclose sensitive data, but COWL would normally prohibit such an operation. In such situations, how can a context *declassify* data, and thus be permitted to send to an arbitrary recipient, or avoid the recipient's being confined?

COWL's *privilege* primitive enables safe declassification. A context may hold one or more privileges, each with respect to some origin. Possession of a privilege for an origin by a context denotes trust that the scripts that execute within that context will not compromise the secrecy of data from that origin. Where might such trust come from (and hence how are privileges granted)? Under the SOP, when a browser retrieves a page from `a.com`, any script within the context for the page is trusted not to violate the secrecy of `a.com`'s data, as these scripts are deemed to be executing on behalf of `a.com`. COWL makes the analogous assumption by granting the privilege for `a.com` to the context that retrieves a page from `a.com`: scripts executing in that context are similarly deemed to be executing on behalf of `a.com`, and thus are trusted not to leak `a.com`'s data to unauthorized parties—even though they can declassify data. Only the COWL runtime can create a new privilege for a valid remote origin upon retrieval of a page from that origin; a script cannot synthesize a privilege for a valid remote origin.

To illustrate the role of privileges in declassification, consider the encrypted Google Docs example application. In the implementation of this application atop COWL, code executing on behalf of `eff.org` (*i.e.,* in a compartment holding the `eff.org` privilege) with a current label $\langle$`eff.org` $\wedge$ `gdoc.com`$\rangle$ is permitted to send messages to a context labeled $\langle$`gdoc.com`$\rangle$. Without the `eff.org` privilege, this flow would not be allowed, as it may leak the EFF's information to Google.

Similarly, code can declassify information when unlabeling messages. Consider now the password checker example application. The left context in Figure 2 leverages its `fb.com` privilege to declassify the password strength result, which is labeled with its origin, to avoid (unecessarily) raising its label to `fb.com`.

COWL generally exercises privileges *implicitly:* if a context holds a privilege, code executing in that context will, with the exception of sending a message, always attempt to use it.[9] COWL, however, lets code control the use of privileges by allowing code to get and set the underlying context's privileges. Code can drop privileges by setting its context's privileges to **`null`**. Dropping privileges is of practical use in confining closely coupled untrusted libraries like jQuery. Setting privileges, on the other hand, increases the trust placed in a context by authorizing it act on behalf of origins. This is especially useful since COWL allows one context to *delegate* its privileges (or a subset of them) to another; this functionality is also instrumental in confining untrusted libraries like jQuery. Finally, COWL also allows a context to create privileges for *fresh* origins, *i.e.,* unique origins that do not have a real protocol (and thus do not map to real servers). These fresh origins are primarily used to *completely* confine a context: the sender can label messages with such an origin, which upon inspection will raise the receiver's label to this "fake" origin, thereby ensuring that it cannot communicate except with the parent (which holds the fresh origin's privilege).

4 APPLICATIONS

In Section 2.2, we characterized four applications and explained why the status-quo web architecture cannot accommodate them satisfactorily. We then described the COWL system's new browser primitives. We now close the loop by demonstrating how to build the aforementioned applications with the COWL primitives.

Encrypted Document Editor The key feature needed by an encrypted document editor is symmetric confinement, where two mutually distrusting scripts can each confine the other's use of data they send one another. Asymmetrically conferring COWL privileges on the distrusting components is the key to realizing this application.

Figure 3 depicts the architecture for an encrypted document editor. The editor has three components: a component which has the user's Google Docs credentials and communicates with the server (`gdoc.com`), the editor proper (also `gdoc.com`), and the component that performs encryption (`eff.org`). COWL provides privacy as follows: if `eff.org` is honest, then COWL ensures that the cleartext of the user's document is not leaked to any origin. If only `gdoc.com` is honest, then `gdoc.com` may be able to recover cleartext (*e.g.,* the encryptor may have used the null "cipher"), but the encryptor should not be able to exfiltrate the cleartext to anyone else.

How does execution of the encrypted document editor proceed? Initially, `gdoc.com` downloads (1) the en-

[8] To continuously process data in "streaming" fashion, one may partition the application into contexts that poll Amazon and Chase's servers for new data and pass labeled responses to the confined context that processes the payloads of the responses.

[9] While the alternative approach of explicit exercise of privileges (*e.g.,* when registering an `onmessage` handler) may be safer [23, 34, 51], we find it a poor fit with existing asynchronous web APIs.

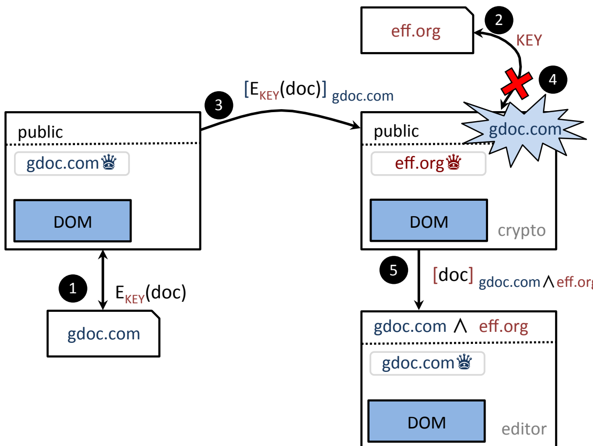


Figure 3: Encrypted document editor architecture.

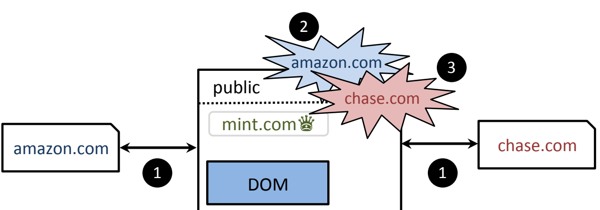


Figure 4: Third-party mashup under COWL.

crypted document from Google's servers. As the document is encrypted, it opens an iframe to `eff.org`, with initial label `public` so it can communicate with the `eff.org` server and download the private key (2) which will be used to decrypt the document. Next, it sends the encrypted document as a labeled Blob, with the label ⟨`gdoc.com`⟩ (3); the iframe unlabels the Blob and raises its label (4) so it can decrypt the document. Finally, the iframe passes the decrypted document (labeled as ⟨`gdoc.com` ∧ `eff.org`⟩) to the iframe (5) implementing the editor proper.

To save the document, these steps proceed in reverse: the editor sends a decrypted document to the encryptor (5), which encrypts it with the private key. Next, the critical step occurs: the encryptor exercises its privileges to send a labeled blob of the encrypted document which is *only* labeled ⟨`gdoc.com`⟩ (3). Since the encryptor is the only compartment with the `eff.org` privilege, all documents must pass through it for encryption before being sent elsewhere; conversely, it itself cannot exfiltrate any data, as it is confined by `gdoc.com` in its label.

We have implemented a password manager atop COWL that lets users safely store passwords on third-party web-accessible storage. We elide its detailed design in the interest of brevity, and note only that it operates similarly to the encrypted document editor.

Third-Party Mashup Labeled XHR as composed with CORS is central to COWL's support for third-party mashups. Today's CORS policies are DAC-only, such that a server must either allow another origin to read its data and fully trust that origin not to disclose the data, or deny the other origin access to the data altogether. Under COWL, however, a server could CORS-whitelist a foreign origin to permit that origin to read its data, and by setting a label on its response, be safe in the knowledge that COWL would appropriately confine the foreign origin's scripts in the browser.

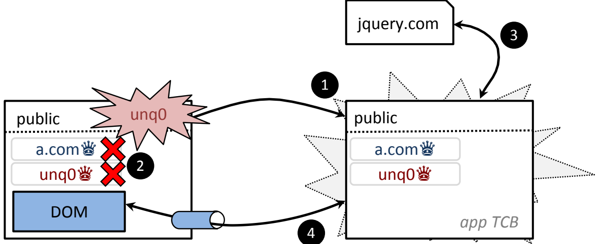


Figure 5: Privilege separation and library confinement.

Figure 4 depicts an application that reconciles a user's Amazon purchases and bank statement. Here, Chase and Amazon respectively expose authenticated read-only APIs for bank statements and purchase histories that whitelist known applications' origins, such as `mint.com`, but set MAC labels on responses.[10] As discussed in Section 7, with MAC in place, COWL allows users to otherwise augment CORS by whitelisting foreign origins on a per-origin basis. The mashup makes requests to both web sites using labeled XHR (1) to receive the bank statement and purchase history as labeled Blobs. Once all of the information is received, the mashup unlabels the responses and raises its context's label accordingly (2–3); doing so restricts communication to the web at large.

Note that in contrast to when solely using CORS, by setting MAC labels on responses, Chase and Amazon need not trust Mint to write bug-free code—COWL confines the Mint code to ensure that it cannot arbitrarily leak sensitive data. As we discuss in Section 7, however, a malicious Mint application could potentially leak data through covert channels. We emphasize that COWL nevertheless offers a significant improvement over the status quo, in which, *e.g.,* users give their login credentials to Mint, and thus not only trust Mint to keep their bank statements confidential, but also not to steal their funds!

Untrusted Third-Party Library COWL can confine tightly coupled untrusted third-party libraries like jQuery by delegating privileges to a trusted context and subsequently dropping them from the main page. In doing so, COWL completely confines the main page, and ensures that it can only communicate with the trusted and unconfined context. Here, the main page may start out with sensitive data in context, or alternatively, receive it from the trusted compartment.

[10] On authentication: note that when the browser sends any XHR (labeled or not) from a foreign origin to origin `chase.com`, it still includes any cookies cached for `chase.com` in the request.

```
interface Label :
  Label Label(String)
  Label and(String or Label)
  Label or(String or Label)
  bool subsumes(Label [,Privilege])
interface Privilege :
  Privilege FreshPrivilege()
  Privilege combine(Privilege)
  readonly attribute Label asLabel
```

(a) Labels and privileges.

```
interface LabeledBlob :
  readonly attribute Label label
  readonly attribute Blob blob
```

(b) Labeled Blobs.

```
interface COWL :
  static void enable()
  static attribute Label label
  static attribute Label clearance
  static attribute Privilege privilege
interface LWorker :
  LWorker LWorker(String, Label
                  [, Privilege, object])
  postMessage(object)
  attribute EventHandler onmessage
```

(c) Labeled compartments.

Figure 6: COWL programming interface in simplified WebIDL.

Figure 5 shows how to use COWL to confine the untrusted jQuery library referenced by a web page. The goal is to establish a separate DOM worker with the `a.com` privilege, while the main browsing context runs jQuery in confined fashion—without privileges or the ability to talk to the network. Initially the main browsing context holds the `a.com` privilege. The page generates a fresh origin `unq0` and spawns a DOM worker (1), delegating it both privileges. The main context then drops its privileges and raises its label to ⟨`unq0`⟩ (2). Finally, the trusted worker downloads jQuery (3) and injects the script content into the main context's DOM (4). When the library is loaded, the main context becomes untrusted, but also fully confined. As the trusted DOM worker holds both privileges, it can freely modify the DOM of the main context, as well as communicate with the wider web. One may view this DOM worker as a *firewall* between the page proper (with the untrusted library) and the rest of the world.

5 Implementation

We implemented COWL in Firefox 31.0a1 and Chromium 31.0.1612.0. Because COWL operates at a context granularity, it admits an implementation as a new DOM-level API for the Gecko and Blink layout engines, without any changes to the browsers' JavaScript engines. Figure 6 shows the core parts of this API. We focus on the Firefox implementation and only describe the Chromium one where the two diverge non-trivially.

Channel	Mechanism
`postMessage`	Cross-compartment wrappers[11]
DOM window properties	Cross-compartment wrappers
Content loading	CSP
XHR	CSP + DOM interposition
Browser storage	SOP + CSP (sandbox)
Other (*e.g.,* iframe height)	DOM interposition

Table 1: Confining code from exfiltrating data using existing browser mechanisms.

5.1 Labeled Browsing Contexts

Gecko's existing isolation model relies on JavaScript compartments, *i.e.,* disjoint JavaScript heaps, both for efficient garbage collection and security isolation [40]. To achieve isolation, Gecko performs all cross-compartment communication (*e.g.,* `postMessage` between iframes) through *wrappers* that implement the object-capability *membrane* pattern [21, 22]; membranes enable sound reasoning about "border crossing" between compartments. Wrappers ensure that an object in one compartment can never directly reference another object in a different compartment. Wrappers also include a security policy, which enforces all inter-compartment access control checks specified by the SOP. Security decisions are made with respect to a compartment's security principal, which contains the origin and CSP of the compartment.

Since COWL's security model is very similar to this existing model, we can leverage these wrappers to introduce COWL's new security policies. We associate a label, clearance, and privilege with each compartment alongside the security principal. Wrappers consider all of these properties together when making security decisions.

Intra-Browser Confinement As shown in Table 1, we rely on wrappers to confine cross-compartment communication. Once confinement mode is enabled, we "recompute" all cross-compartment wrappers to use our MAC wrapper policy and thereby ensure that all subsequent cross-compartment access is mediated not only by the SOP, but also by confinement. For `postMessage`, our policy ensures that the receiver's label subsumes that of the sender (taking the receiver's privileges into consideration); otherwise the message is silently dropped. For a cross-compartment DOM property access, we additionally check that the sender's label subsumes that of the receiver—*i.e.,* that the labels of the compartments are equivalent after considering the sender's privileges (in addition to the same-origin check performed by the SOP).

Blink's execution contexts (the dual to Gecko's compartments) do not rely on wrappers to enforce cross-context access control. Instead, Blink implements the

[11] Since the Chromium architecture does not have cross-compartment wrappers, we modify the DOM binding code to insert label checks.

SOP security checks in the DOM binding code for a limited subset of DOM elements that may allow cross-origin access. Since COWL policies are more fine-grained, we modified the binding code to extend the security checks to all DOM objects and also perform label checks when confinement mode is enabled. Unfortunately, without wrappers, shared references cannot efficiently be revoked (*i.e.,* without walking the heap). Hence, before enabling confinement mode, a page can create a same-origin iframe with which it shares references, and the iframe can thereafter leak any data from the parent even if the latter's label is raised. To prevent this eventuality, our current Chromium API allows senders to disallow unlabeling Blobs if the target created any children before entering confinement mode.

Our implementations of LWorkers, whose API appears in Figure 6c, reuse labeled contexts straightforwardly. In fact, the `LWorker` constructor simply creates a new compartment with a fresh origin that contains a fresh JavaScript global object to which we attach the XHR constructor, COWL API, and primitives for communicating with the parent (*e.g.,* `postMessage`). Since LWorkers may have access to their parents' DOM, however, our wrappers distinguish them from other contexts to bypass SOP checks and only restrict DOM access according to MAC. This implementation is very similar to the content scripts used by Chrome and Firefox extensions [10, 26].

Browser-Server Confinement As shown in Table 1, we confine external communication (including XHR, content loading, and navigation) using CSP. While CSP alone is insufficient for providing flexible confinement,[12] it sufficiently addresses our external communication concern by precisely controlling from where a page loads content, performs XHR requests to, *etc.* To this end, we set a custom CSP policy whenever the compartment label changes, *e.g.,* with `COWL.label`. For instance, if the effective compartment label is `Label("https://bank.ch").and("https://amazon.com")`, all the underlying CSP directives are set to `'none'` (*e.g.,* `default-src 'none'`), disallowing all network communication. We also disable navigation with the `'sandbox'` directive [46–48].

Browser Storage Confinement As shown in Table 1, we use the `sandbox` directive to restrict access to storage (*e.g.,* cookies and HTML5 local storage [47]), as have other systems [5]. We leave the implementation of labeled storage as future work.

6 Evaluation

Performance largely determines acceptance of new browser features in practice. We evaluate the performance of COWL by measuring the cost of our new primitives as well as their impact on legacy web sites that do not use COWL's features. Our experiments consist of micro-benchmarks of API functions and end-to-end benchmarks of our example applications. We conducted all measurements on a 4-core i7-2620M machine with 16GB of RAM running GNU/Linux 3.13. The browser retrieved applications from the Node.js web server over the loopback interface. We note that these measurements are harsh for COWL, in that they omit network latency and the complex intra-context computation and DOM rendering of real-world applications, all of which would mask COWL's overhead further. Our key findings include:

- COWL's latency impact on legacy sites is negligible.
- Confining code with LWorkers is inexpensive, especially when compared to iframes/Workers. Indeed, the performance of our end-to-end confined password checker is only 5 ms slower than that of an inlined `script` version.
- COWL's incurs low overhead when enforcing confinement on mashups. The greatest overhead observed is 16% (for the encrypted document editor). Again, the absolute slowdown of 16 ms is imperceptible by users.

[12] There are two primary reasons. First, JavaScript code cannot (yet) modify a page's CSP. And, second, CSP does not (yet) provide a directive for restricting in-browser communication, *e.g.,* with `postMessage`.

	Firefox			Chromium		
	vanilla	unlabeled	labeled	vanilla	unlabeled	labeled
New iframe	14.4	14.5	14.4	50.6	48.7	51.8
New worker	15.9	15.4	0.9†	18.9	18.9	3.3†
Iframe comm.	0.11	0.11	0.12	0.04	0.04	0.04
XHR comm	3.5	3.6	3.7	7.0	7.4	7.2
Worker comm.	0.20	0.24	0.03‡	0.07	0.07	0.03‡

Table 2: Micro-benchmarks, in milliseconds.

6.1 Micro-Benchmarks

Context Creation Table 2 shows micro-benchmarks for the stock browsers (vanilla), the COWL browsers with confinement mode turned off (unlabeled), and with confinement mode enabled (labeled). COWL adds negligible latency to compartment creation; indeed, except for LWorkers (†), the differences in creation times are of the order of measurement variability. We omit measurements of labeled "normal" Workers since they do not differ from those of unlabeled Workers. We attribute COWL's iframe-creation speedup in Chromium to measurement variability. We note that the cost of creating LWorkers is considerably less than that for "normal" Workers, which run in separate OS threads (†).

Communication The iframe, worker, and XHR communication measurements evaluate the round-trip latencies across iframes, workers, and the network. For the XHR benchmark, we report the cost of using the labeled XHR constructor averaged over 10,000 requests. Our

Chromium implementation uses an LWorker to wrap the unmodified XHR constructor, so the cost of labeled XHR incorporates an additional cross-context call. As with creation, communicating with LWorkers (‡) is considerably faster than with "normal" Workers. This speedup arises because a lightweight LWorker shares an OS thread and event loop with their parent.

Labels We measured the cost of setting/getting the current label and the average cost of a label check in Firefox. For a randomly generated label with a handful of origins, these operations take on the order of one microsecond. The primary cost is recomputing cross-compartment wrappers and the underlying CSP policy, which ends up costing up to 13ms (*e.g.,* when the label is raised from public to a third-party origin). For many real applications, we expect raising the current label to be a rare occurrence. Moreover, there is much room for optimization (*e.g.,* porting COWL to the newest CSP implementation, which sets policies 15× faster [19]).

DOM We also executed the Dromaeo benchmark suite [29], which evaluates the performance of core functionality such as querying, traversing, and manipulating the DOM, in Firefox and Chromium. We found the performance of the vanilla and unlabeled browsers to be on par: the greatest slowdown was under 4%.

6.2 End-to-End Benchmarks

To focus on measuring COWL's overhead, we compare our apps against similarly compartmentalized but non-secure apps—*i.e.,* apps that perform no security checks.

Password-Strength Checker We measure the average duration of creating a new LWorker, fetching an 8 KB checker script based on [24], and checking a password sixteen characters in length. The checker takes an average of 18 ms (averaged over ten runs) on Firefox (labeled), 4 ms less than using a Worker on vanilla Firefox. Similarly, the checker running on labeled Chromium is 5 ms faster than the vanilla counterpart (measured at 54 ms). In both cases COWL achieves a speedup because its LWorkers are cheaper than normal Workers. However, these measurements are roughly 5 ms slower than simply loading the checker using an unsafe `script` tag.

Encrypted Document Editor We measure the end-to-end time taken to load the application and encrypt a 4 KB document using the SJCL AES-128 library [32]. The total run time includes the time taken to load the document editor page, which in turn loads the encryption-layer iframe, which further loads the editor proper. On Firefox (labeled) the workload completes in 116 ms; on vanilla Firefox, a simplified and unconfined version completes in 100ms. On Chromium, the performance measurements were comparable; the completion time was within 1ms of 244ms. The most expensive operation in the COWL-enabled Firefox app is raising the current label, since it requires changing the underlying document origin and recomputing the cross-compartment wrappers and CSP.

Third-Party Mashup We implemented a very simple third-party mashup application that makes a labeled XHR request to two unaffiliated origins, each of which produces a response containing a 27-byte JSON object with a numerical property, and sums the responses together. The corresponding vanilla app is identical, but uses the normal XHR object. In both cases we use CORS to permit cross-origin access. The Firefox (labeled) workload completes in 41 ms, which is 6 ms slower than the vanilla version. As in the document editor the slowdown derives from raising the current label, though in this case only for a single iframe. On Chromium (labeled) the workload completes in 55 ms, 2 ms slower than the vanilla one; the main slowdown here derives from our implementing labeled XHR with a wrapping LWorker.

Untrusted Third-Party Library We measured the load time of a banking application that incorporates jQuery and a library that traverses the DOM to replace phone numbers with links. The latter library uses XHR in attempt to leak the page's content. We compartmentalize the main page into a public outer component and a sensitive iframe containing the bank statement. In both compartments, we place the bank's trusted code (which loads the libraries) in a trusted labeled DOM worker with access to the page's DOM. We treat the rest of the code as untrusted. As our current Chromium implementation does not yet support DOM access for LWorkers, we only report measurements for Firefox. The measured latency on Firefox (labeled) is 165 ms, a 5 ms slowdown when compared to the unconfined version running on vanilla Firefox. Again, COWL prevents sensitive content from being exfiltrated and incurs negligible slowdown.

7 Discussion and Limitations

We now discuss the implications of certain facets of COWL's design, and limitations of the system.

User-Configured Confinement Recall that in the status-quo web security architecture, to allow cross-origin sharing, a server must grant individual foreign origins access to its data with CORS in an all-or-nothing, DAC fashion. COWL improves this state of affairs by allowing a COWL-aware server to more finely restrict how its shared data is disseminated—*i.e.,* when the server grants a foreign origin access to its data, it can confine the foreign origin's script(s) by setting a label on responses it sends the client.

Unfortunately, absent a permissive CORS header that whitelists the origins of applications that a user wishes to use, the SOP prohibits foreign origins from reading responses from the server, even in a COWL-enabled browser. Since a server's operator may not be aware of all applications its users may wish to use, the result is

usually the same status-quo unpalatable choice between functionality and privacy—*e.g.,* give one's bank login credentials to Mint, or one cannot use the Mint application. For this reason, our COWL implementation lets browser users augment CORS by configuring for an origin (*e.g.,* `chase.com`) any foreign origins (*e.g.,* `mint.com`, `benjamins.biz`) they wish to additionally whitelist. In turn, COWL will confine these client-whitelisted origins (*e.g.,* `mint.com`) by labeling every response from the configured origin (`chase.com`). COWL obeys the server-supplied label when available and server whitelisting is *not* provided. Otherwise, COWL conservatively labels the response with a *fresh* origin (as described in Section 3.3). The latter ensures that once the response has been inspected, the code cannot communicate with *any* server, including at the *same* origin, since such requests carry the risks of self-exfiltration [11] and cross-site request forgery [39].

Covert Channels In an ideal confinement system, it would always be safe to let untrusted code compute on sensitive data. Unfortunately, real-world systems such as browsers typically exhibit *covert* channels that malicious code may exploit to exfiltrate sensitive data. Since COWL extends existing browsers, we do not protect against covert channel attacks. Indeed, malicious code can leverage covert channels already present in today's browsers to leak sensitive information. For instance, a malicious script within a confined context may be able to modulate sensitive data by varying rendering durations. A less confined context may then in turn exfiltrate the data to a remote host [20]. It is important to note, however, that COWL does not introduce new covert channels—our implementations re-purpose existing (software-based) browser isolation mechanisms (V8 contexts and SpiderMonkey compartments) to enforce MAC policies. Moreover, these MAC policies are generally more restricting than existing browser policies: they prevent unauthorized data exfiltration through *overt* channels and, in effect, force malicious code to resort to using covert channels.

The only fashion in which COWL relaxes status-quo browser policies is by allowing users to override CORS to permit cross-origin (labeled) sharing. Does this functionality introduce new risks? Whitelisting is user controlled (*e.g.,* the user must explicitly allow `mint.com` to read `amazon.com` and `chase.com` data), and code reading cross-origin data is subject to MAC (*e.g.,* `mint.com` cannot arbitrarily exfiltrate the `amazon.com` or `chase.com` data after reading it). In contrast, today's mashups like `mint.com` ask users for their passwords. COWL is strictly an improvement: under COWL, when a user decides to trust a mashup integrator such as `mint.com`, she *only* trusts the app to not leak her data through covert channels. Nevertheless, users can make poor security choices. Whitelisting malicious origins would be no exception; we recognize this as a limitation of COWL that must be communicated to the end-user.

A trustworthy developer can leverage COWL's support for *clearance* when compartmentalizing his application to ensure that only code that actually relies on cross-origin data has access to it. Clearance is a label that serves as an upper bound on a context's current label. Since COWL ensures that the current label is adjusted according to the sensitivity of the data being read, code cannot read (and thus leak) data labeled above the clearance. Thus, Mint can assign a "low" clearance to untrusted third-party libraries, *e.g.,* to keep `chase.com`'s data confidential. These libraries will then not be able to leak such data through covert channels, even if they are malicious.

Expressivity of Label Model COWL uses DC labels [33] to enforce confinement according to an information flow control discipline. Although this approach captures a wide set of confinement policies, it is not expressive enough to handle policies with a circular flow of information [6] or some policies expressible in more powerful logics (*e.g.,* first order logic, as used by Nexus [30]). DC labels are, however, as expressive as other popular label models [25], including Myers and Liskov's Decentralized Label Model [27]. Our experience implementing security policies with them thus far suggests they are expressive enough to support featureful web applications.

We adopted DC labels largely because their fit with web origins pays practical dividends. First, as developers already typically express policies by whitelisting origins, we believe they will find DC labels intuitive to use. Second, because both DC labels and today's web policies are defined in terms of origins, the implementation of COWL can straightforwardly reuse the implementation of existing security mechanisms, such as CSP.

8 Related Work

Existing browser confinement systems based on information flow control can be classified either as *fine-grained* or *coarse-grained*. The former associate IFC policies with individual objects, while the latter associate policies with entire browsing contexts. We compare COWL to previously proposed systems in both categories, then contrast the two categories' overall characteristics.

Coarse-grained IFC COWL shares many features with existing coarse-grained systems. BFlow [50], for example, allows web sites to enforce confinement policies stricter than the SOP via *protection zones*—groups of iframes sharing a common label. However, BFlow cannot mediate between mutually distrustful principals—*e.g.,* the encrypted document editor is not directly implementable with BFlow. This is because only asymmetric confinement is supported—a sub-frame cannot impose any restrictions on its parent. For the same reasons, BFlow cannot support applications that require security policies more flexible

than the SOP, such as our third-party mashup example. These differences reflect different goals for the two systems. BFlow's authors set out to confine untrusted third-party scripts, while we also seek to support applications that incorporate code from mutually distrusting parties.

More recently, Akhawe *et al.* propose the data-confined sandbox (DCS) system [5], which allows pages to intercept and monitor the network, storage, and cross-origin channels of `data:` URI iframes. The limitation to `data:` URI iframes means DCS cannot confine the common case of a service provided in an iframe [31]. Like BFlow, DCS does not offer symmetric confinement, and does not incorporate functionality to let developers build applications like third-party mashups.

Fine-grained IFC Per-object-granularity IFC makes it easier to confine untrusted libraries that are closely coupled with trusted code on a page (*e.g.,* jQuery) and avoid the problem of *over-tainting,* where a single context accumulates taint as it inspects more data.

JSFlow [15] is one such fine-grained JavaScript IFC system, which enforces policies by executing JavaScript in an interpreter written in JavaScript. This approach incurs a two order of magnitude slowdown. JSFlow's authors suggest that this cost makes JSFlow a better fit for use as a development tool than as an "always-on" privacy system for users' browsers. Additionally, JSFlow does not support applications that rely on policies more flexible than the SOP, such as our third-party mashup example.

The FlowFox fine-grained IFC system [12] enforces policies with secure-multi execution (SME) [13]. SME ensures that no leaks from a sensitive context can leak into a less sensitive context by executing a program multiple times. Unlike JSFlow and COWL, SME is not amenable to scenarios where declassification plays a key role (*e.g.,* the encrypted editor or the password manager). FlowFox's labeling of user interactions and metadata (history, screen size, *etc.*) do allow it to mitigate history sniffing and behavior tracking; COWL does not address these attacks.

While fine-grained IFC systems may be more convenient for developers, they impose new language semantics for developers to learn, require invasive modifications to the JavaScript engine, and incur greater performance overhead. In contrast, because COWL repurposes familiar isolation constructs and does not require JavaScript engine modifications, it is relatively straightforward to add to legacy browsers. It also only adds overhead to cross-compartment operations, rather than to all JavaScript execution. The typically short lifetime of a browsing context helps avoid excessive accumulation of taint. We conjecture that coarse-grained and fine-grained IFC are equally expressive, provided one may use arbitrarily many compartments—a cost in programmer convenience. Finally, coarse- and fine-grained mechanisms are not mutually exclusive. For instance, to confine legacy (non-compartmentalized) JavaScript code, one could deploy JSFlow within a COWL context.

Sandboxing The literature on sandboxing and secure subsets of JavaScript is rich, and includes Caja [1], BrowserShield [28], WebJail [37], TreeHouse [18], JSand [4], SafeScript [36], Defensive JavaScript [9], and Embassies [16]). While our design has been inspired by some of these systems (*e.g.,* TreeHouse), the usual goals of these systems are to mediate security-critical operations, restrict access to the DOM, and restrict communication APIs. In contrast to the mandatory nature of confinement, however, these systems impose most restrictions in discretionary fashion, and are thus not suitable for building some of the applications we consider (in particular, the encrypted editor). Nevertheless, we believe that access control and language subsets are crucial complements to confinement for building robustly secure applications.

9 Conclusion

Web applications routinely pull together JavaScript contributed by parties untrusted by the user, as well as by mutually distrusting parties. The lack of confinement for untrusted code in the status-quo browser security architecture puts users' privacy at risk. In this paper, we have presented COWL, a label-based MAC system for web browsers that preserves users' privacy in the common case where untrusted code computes over sensitive data. COWL affords developers flexibility in synthesizing web applications out of untrusted code and services while preserving users' privacy. Our positive experience building four web applications atop COWL for which privacy had previously been unattainable in status-quo web browsers suggests that COWL holds promise as a practical platform for preserving privacy in today's pastiche-like web applications. And our measurements of COWL's performance overhead in the Firefox and Chromium browsers suggest that COWL's privacy benefits come at negligible end-to-end cost in performance.

Acknowledgements

We thank Bobby Holley, Blake Kaplan, Ian Melven, Garret Robinson, Brian Smith, and Boris Zbarsky for helpful discussions of the design and implementation of COWL. We thank Stefan Heule and John Mitchell for useful comments on formal aspects of the design. And we thank our shepherd Mihai Budiu and the anonymous reviewers for their helpful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by the EPSRC under grant EP/K032542/1, the Swedish research agencies VR and STINT, the Barbro Osher Pro Suecia foundation, and by multiple gifts from Google (to Stanford and UCL). Deian Stefan and Edward Z. Yang are supported through the NDSEG Fellowship Program.

References

[1] Google Caja. A source-to-source translator for securing JavaScript-based web content. http://code.google.com/p/google-caja/, 2013.

[2] Mint. http://www.mint.com/, 2013.

[3] jQuery Usage Statistics: Websites using jQuery. http://trends.builtwith.com/javascript/jQuery, 2014.

[4] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.

[5] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song. Data-confined HTML5 applications. In *ESORICS*, 2013.

[6] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *Security and Privacy*, 1995.

[7] A. Barth. The web origin concept. Technical report, IETF, 2011. URL https://tools.ietf.org/html/rfc6454.

[8] A. Barth, C. Jackson, and J. Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[9] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security*, 2013.

[10] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the Google Chrome extension security architecture. In *USENIX Security*, 2012.

[11] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. In *Web 2.0 Security and Privacy*, 2012.

[12] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *CCS*, 2012.

[13] D. Devriese and F. Piessens. Noninterference through Secure Multi-Execution. In *Security and Privacy*, 2010.

[14] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *OSDI*, 2005.

[15] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*, 2014.

[16] J. Howell, B. Parno, and J. R. Douceur. Embassies: Radically refactoring the Web. In *NSDI*, 2013.

[17] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your ifcexception are belong to us. In *Security and Privacy*, 2013.

[18] L. Ingram and M. Walfish. Treehouse: JavaScript sandboxes to help web developers help themselves. In *USENIX ATC*, 2012.

[19] C. Kerschbaumer. Faster Content Security Policy (CSP). https://blog.mozilla.org/security/2014/09/10/faster-csp/, 2014.

[20] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In *CCS*, 2013.

[21] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[22] M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *ASIAN*, 2003.

[23] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://zesty.ca/capmyths/usenix.pdf.

[24] S. Moitozo. http://www.geekwisdom.com/js/passwordmeter.js, 2006.

[25] B. Montagu, B. C. Pierce, and R. Pollack. A theory of information-flow labels. In *CSF*, June 2013.

[26] Mozilla. Add-on builder and SDK. https://addons.mozilla.org/en-US/developers/docs/sdk/, 2013.

[27] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *TOSEM*, 9(4), 2000.

[28] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic HTML. *TWEB*, 1(3), Sept. 2007.

[29] J. Reisg. Dromaeo: JavaScript performance testing. http://dromaeo.com/, 2014.

[30] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *SOSP*, 2011.

[31] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In *NDSS*, 2013.

[32] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in JavaScript. In *ACSAC*, 2009.

[33] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *NordSec*, 2011.

[34] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, 2011.

[35] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, 2012.

[36] M. Ter Louw, P. H. Phung, R. Krishnamurti, and V. N. Venkatakrishnan. SafeScript: JavaScript transformation for policy enforcement. In *Secure IT Systems*, 2013.

[37] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. In *ACSAC*, 2011.

[38] A. Van Kesteren. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`, 2012.

[39] B. Vibber. CSRF token-stealing attack (user.tokens). `https://bugzilla.wikimedia.org/show_bug.cgi?id=34907`, 2014.

[40] G. Wagner, A. Gal, C. Wimmer, B. Eich, and M. Franz. Compartmental memory management in a modern web browser. *SIGPLAN Notices*, 46 (11), 2011.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. *ACM SIGOPS Operating Systems Review*, 41(6), 2007.

[42] WC3. Content Security Policy 1.0. `http://www.w3.org/TR/CSP/`, 2012.

[43] WC3. HTML5 web messaging. `http://www.w3.org/TR/webmessaging/`, 2012.

[44] WC3. Web Workers. `http://www.w3.org/TR/workers/`, 2012.

[45] WC3. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`, 2013.

[46] WC3. Content Security Policy 1.1. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`, 2013.

[47] WC3. HTML5. `http://www.w3.org/TR/html5/`, 2013.

[48] WHATWG. HTML living standard. `http://developers.whatwg.org/`, 2013.

[49] E. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp. Toward principled browser security. In *HotOS*, 2013.

[50] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys*, 2009.

[51] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[52] M. Zelwski. Browser security handbook, part 2. `HTtp://code.google.com/p/browsersec/wiki/Part2`, 2011.

Code-Pointer Integrity

Volodymyr Kuznetsov*, László Szekeres‡, Mathias Payer†,§
George Candea*, R. Sekar‡, Dawn Song†

*École Polytechnique Fédérale de Lausanne (EPFL),
†UC Berkeley, ‡Stony Brook University, §Purdue University

Abstract

Systems code is often written in low-level languages like C/C++, which offer many benefits but also delegate memory management to programmers. This invites memory safety bugs that attackers can exploit to divert control flow and compromise the system. Deployed defense mechanisms (e.g., ASLR, DEP) are incomplete, and stronger defense mechanisms (e.g., CFI) often have high overhead and limited guarantees [19, 15, 9].

We introduce code-pointer integrity (CPI), a new design point that *guarantees the integrity of all code pointers* in a program (e.g., function pointers, saved return addresses) and thereby prevents all control-flow hijack attacks, including return-oriented programming. We also introduce code-pointer separation (CPS), a relaxation of CPI with better performance properties. CPI and CPS offer substantially better security-to-overhead ratios than the state of the art, they are practical (we protect a complete FreeBSD system and over 100 packages like apache and postgresql), effective (prevent all attacks in the RIPE benchmark), and efficient: on SPEC CPU2006, CPS averages 1.2% overhead for C and 1.9% for C/C++, while CPI's overhead is 2.9% for C and 8.4% for C/C++.

A prototype implementation of CPI and CPS can be obtained from `http://levee.epfl.ch`.

1 Introduction

Systems code is often written in memory-unsafe languages; this makes it prone to memory errors that are the primary attack vector to subvert systems. Attackers exploit bugs, such as buffer overflows and use after free errors, to cause memory corruption that enables them to steal sensitive data or execute code that gives them control over a remote system [44, 37, 12, 8].

Our goal is to secure systems code against all *control-flow hijack* attacks, which is how attackers gain remote control of victim systems. Low-level languages like C/C++ offer many benefits to system programmers, and we want to make these languages safe to use while preserving their benefits, not the least of which is performance. Before expecting any security guarantees from systems we must first secure their building blocks.

There exist a few protection mechanism that can reduce the risk of control-flow hijack attacks without imposing undue overheads. Data Execution Prevention (DEP) [48] uses memory page protection to prevent the introduction of new executable code into a running application. Unfortunately, DEP is defeated by code reuse attacks, such as return-to-libc [37] and return oriented programming (ROP) [44, 8], which can construct arbitrary Turing-complete computations by chaining together existing code fragments of the original application. Address Space Layout Randomization (ASLR) [40] places code and data segments at random addresses, making it harder for attackers to reuse existing code for execution. Alas, ASLR is defeated by pointer leaks, side channels attacks [22], and just-in-time code reuse attacks [45]. Finally, stack cookies [14] protect return addresses on the stack, but only against continuous buffer overflows.

Many defenses can improve upon these shortcomings but have not seen wide adoption because of the overheads they impose. According to a recent survey [46], these solutions are incomplete and bypassable via sophisticated attacks and/or require source code modifications and/or incur high performance overhead. These approaches typically employ language modifications [25, 36], compiler modifications [13, 3, 17, 34, 43], or rewrite machine code binaries [38, 54, 53]. Control-flow integrity protection (CFI) [1, 29, 53, 54, 39], a widely studied technique for practical protection against control-flow hijack attacks, was recently demonstrated to be ineffective [19, 15, 9].

Existing techniques cannot both *guarantee protection* against control-flow hijacks and impose *low overhead* and *no changes* to how the programmer writes code. For example, memory-safe languages guarantee that a memory object can only be accessed using pointers properly based on that specific object, which in turn makes control-flow hijacks impossible, but this approach requires runtime checks to verify the temporal and spatial correctness of pointer computations, which inevitably induces undue overhead, especially when retrofitted to memory-unsafe languages. For example, state-of-the-art memory safety implementations for C/C+ incur $\geq 2\times$

overhead [35]. We observe that, in order to render control-flow hijacks impossible, it is sufficient to guarantee the integrity of code pointers, i.e., those that are used to determine the targets of indirect control-flow transfers (indirect calls, indirect jumps, or returns).

This paper introduces code-pointer integrity (CPI), a way to enforce precise, deterministic memory safety for all code pointers in a program. The key idea is to split process memory into a *safe region* and a *regular region*. CPI uses static analysis to identify the set of memory objects that must be protected in order to guarantee memory safety for code pointers. This set includes all memory objects that contain code pointers and all data pointers used to access code pointers indirectly. All objects in the set are then stored in the safe region, and the region is isolated from the rest of the address space (e.g., via hardware protection). The safe region can only be accessed via memory operations that are proven at compile time to be safe or that are safety-checked at runtime. The regular region is just like normal process memory: it can be accessed without runtime checks and, thus, with no overhead. In typical programs, the accesses to the safe region represent only a small fraction of all memory accesses (6.5% of all pointer operations in SPEC CPU2006 need protection). Existing memory safety techniques cannot efficiently protect only a subset of memory objects in a program, rather they require instrumenting *all* potentially dangerous pointer operations.

CPI fully protects the program against all control-flow hijack attacks that exploit program memory bugs. CPI requires no changes to how programmers write code, since it automatically instruments pointer accesses at compile time. CPI achieves low overhead by selectively instrumenting only those pointer accesses that are necessary and sufficient to formally guarantee the integrity of all code pointers. The CPI approach can also be used for data, e.g., to selectively protect sensitive information like the process UIDs in a kernel.

We also introduce code-pointer separation (CPS), a relaxed variant of CPI that is better suited for code with abundant virtual function pointers. In CPS, all code pointers are placed in the safe region, but pointers used to access code pointers indirectly are left in the regular region (such as pointers to C++ objects that contain virtual functions). Unlike CPI, CPS may allow certain control-flow hijack attacks, but it still offers stronger guarantees than CFI and incurs negligible overhead.

Our experimental evaluation shows that our proposed approach imposes sufficiently low overhead to be deployable in production. For example, CPS incurs an average overhead of 1.2% on the C programs in SPEC CPU2006 and 1.9% for all C/C++ programs. CPI incurs on average 2.9% overhead for the C programs and 8.4% across all C/C++ SPEC CPU2006 programs. CPI and CPS are effective: they prevent 100% of the attacks in the RIPE benchmark and the recent attacks [19, 15, 9] that bypass CFI, ASLR, DEP, and all other Microsoft Windows protections. We compile and run with CPI/CPS a complete FreeBSD distribution along with ≥ 100 widely used packages, demonstrating that the approach is practical. This paper makes the following contributions:

1. Definition of two new program properties that offer a security-benefit to enforcement-cost ratio superior to the state of the art: code-pointer integrity (CPI) guarantees control flow cannot be hijacked via memory bugs, and code-pointer separation (CPS) provides stronger security guarantees than control-flow integrity but at negligible cost.
2. An efficient compiler-based implementation of CPI and CPS for unmodified C/C++ code.
3. The first practical and complete OS distribution (based on FreeBSD) with full protection built-in against control-flow hijack attacks.

In the rest of the paper we introduce our threat model (§2), describe CPI and CPS (§3), present our implementation (§4), evaluate our approach (§5), discuss related work (§6), and conclude (§7). We formalize the CPI enforcement mechanism and provide a sketch of its correctness proof in Appendix A.

2 Threat Model

This paper is concerned solely with control-flow hijack attacks, namely ones that give the attacker control of the instruction pointer. The purpose of this type of attack is to divert control flow to a location that would not otherwise be reachable in that same context, had the program not been compromised. Examples of such attacks include forcing a program to jump (i) to a location where the attacker injected shell code, (ii) to the start of a chain of return-oriented program fragments ("gadgets"), or (iii) to a function that performs an undesirable action in the given context, such as calling `system()` with attacker-supplied arguments. Data-only attacks, i.e., that modify or leak unprotected non-control data, are out of scope.

We assume powerful yet realistic attacker capabilities: full control over process memory, but no ability to modify the code segment. Attackers can carry out arbitrary memory reads and writes by exploiting input-controlled memory corruption errors in the program. They cannot modify the code segment, because the corresponding pages are marked read-executable and not writable, and they cannot control the program loading process. These assumptions ensure the integrity of the original program code instrumented at compile time, and enable the program loader to safely set up the isolation between the safe and regular memory regions. Our assumptions are consistent with prior work in this area.

3 Design

We now present the terminology used to describe our design, then define the code-pointer integrity property (§3.1), describe the corresponding enforcement mechanism (§3.2), and define a relaxed version that trades some security guarantees for performance (§3.3). We further formalize the CPI enforcement mechanism and sketch its correctness proof in Appendix A.

We say a pointer dereference is *safe* iff the memory it accesses lies within the target object on which the dereferenced pointer is based. A *target object* can either be a memory object or a control flow destination. By *pointer dereference* we mean accessing the memory targeted by the pointer, either to read/write it (for data pointers) or to transfer control flow to its location (for code pointers). A *memory object* is a language-specific unit of memory allocation, such as a global or local variable, a dynamically allocated memory block, or a sub-object of a larger memory object (e.g., a field in a `struct`). Memory objects can also be program-specific, e.g., when using custom memory allocators. A *control flow destination* is a location in the code, such as the start of a function or a return location. A target object always has a well defined lifetime; for example, freeing an array and allocating a new one with the same address creates a different object.

We say a pointer is *based on* a target object X iff the pointer is obtained at runtime by (i) allocating X on the heap, (ii) explicitly taking the address of X, if X is allocated statically, such as a local or global variable, or is a control flow target (including return locations, whose addresses are implicitly taken and stored on the stack when calling a function), (iii) taking the address of a sub-object y of X (e.g., a field in the X `struct`), or (iv) computing a pointer expression (e.g., pointer arithmetic, array indexing, or simply copying a pointer) involving operands that are either themselves based on object X or are not pointers. This is slightly stricter version of C99's "based on" definition: we ensure that each pointer is based on at most one object.

The execution of a program is *memory-safe* iff all pointer dereferences in the execution are safe. A program is memory-safe iff all its possible executions (for all inputs) are memory-safe. This definition is consistent with the state of the art for C/C++, such as SoftBounds+CETS [34, 35]. Precise memory safety enforcement [34, 36, 25] tracks the based-on information for each pointer in a program, to check the safety of each pointer dereference according to the definition above; the detection of an unsafe dereference aborts the program.

3.1 The Code-Pointer Integrity (CPI) Property

A program execution satisfies the code-pointer integrity property iff all its dereferences that either dereference or access sensitive pointers are safe. *Sensitive pointers* are code pointers and pointers that may later be used to access sensitive pointers. Note that the sensitive pointer definition is recursive, as illustrated in Fig. 1. According to case (iv) of the based-on definition above, dereferencing a pointer to a pointer will correspondingly propagate the based-on information; e.g., an expression `*p = &q` copies the result of `&q`, which is a pointer based on `q`, to a location pointed to by `p`, and associates the based-on metadata with that location. Hence, the integrity of the based-on metadata associated with sensitive pointers requires that pointers used to update sensitive pointers be sensitive as well (we discuss implications of relaxing this definition in §3.3). The notion of a sensitive pointer is dynamic. For example, a `void*` pointer 2 in Fig. 1 is sensitive when it points at another sensitive pointer at run time, but it is not sensitive when it points to an integer.

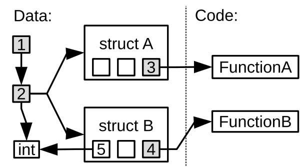


Figure 1: CPI protects code pointers 3 and 4 and pointers 1 and 2 (which may access pointers 3 and 4 indirectly). Pointer 2 of type `void*` may point to different objects at different times. The `int*` pointer 5 and non-pointer data locations are not protected.

A memory-safe program execution trivially satisfies the CPI property, but memory-safety instrumentation typically has high runtime overhead, e.g., $\geq 2\times$ in state-of-the-art implementations [35]. Our observation is that only a small subset of all pointers are responsible for making control-flow transfers, and so, by enforcing memory safety only for control-sensitive data (and thus incurring no overhead for all other data), we obtain important security guarantees while keeping the cost of enforcement low. This is analogous to the control-plane/data-plane separation in network routers and modern servers [5], with CPI ensuring the safety of data that influences, directly or indirectly, the control plane.

Determining precisely the set of pointers that are sensitive can only be done at run time. However, the CPI property can still be enforced using any over-approximation of this set, and such over-approximations can be obtained at compile time, using static analysis.

3.2 The CPI Enforcement Mechanism

We now describe a way to retrofit the CPI property into a program P using a combination of static instrumentation and runtime support. Our approach consists of a *static analysis* pass that identifies all sensitive pointers in P and all instructions that operate on them (§3.2.1), an *instrumentation* pass that rewrites P to "protect" all sensitive pointers, i.e., store them in a separate, safe memory region and associate, propagate, and check their based-

on metadata (§3.2.2), and an instruction-level *isolation* mechanism that prevents non-protected memory operations from accessing the safe region (§3.2.3). For performance reasons, we handle return addresses stored on the stack separately from the rest of the code pointers using a *safe stack* mechanism (§3.2.4).

3.2.1 CPI Static Analysis

We determine the set of sensitive pointers using type-based static analysis: a pointer is sensitive if its type is sensitive. Sensitive types are: pointers to functions, pointers to sensitive types, pointers to composite types (such as struct or array) that contains one or more members of sensitive types, or universal pointers (i.e., void*, char* and opaque pointers to forward-declared structs or classes). A programmer could additionally indicate, if desired, other types to be considered sensitive, such as struct ucred used in the FreeBSD kernel to store process UIDs and jail information. All code pointers that a compiler or runtime creates implicitly (such as return addresses, C++ virtual table pointers, and setjmp buffers) are sensitive as well.

Once the set of sensitive pointers is determined, we use static analysis to find all program instructions that manipulate these pointers. These instructions include pointer dereferences, pointer arithmetic, and memory (de-)allocation operations that calls to either (i) corresponding standard library functions, (ii) C++ new/delete operators, or (iii) manually annotated custom allocators.

The derived set of sensitive pointers is over-approximate: it may include universal pointers that never end up pointing to sensitive values at runtime. For instance, the C/C++ standard allows char* pointers to point to objects of any type, but such pointers are also used for C strings. As a heuristic, we assume that char* pointers that are passed to the standard libc string manipulation functions or that are assigned to point to string constants are not universal. Neither the over-approximation nor the char* heuristic affect the security guarantees provided by CPI: over-approximation merely introduces extra overhead, while heuristic errors may result in false violation reports (though we never observed any in practice).

Memory manipulation functions from libc, such as memset or memcpy, could introduce a lot of overhead in CPI: they take void* arguments, so a libc compiled with CPI would instrument all accesses inside the functions, regardless of whether they are operating on sensitive data or not. CPI's static analysis instead detects such cases by analyzing the real types of the arguments prior to being cast to void*, and the subsequent instrumentation pass handles them separately using type-specific versions of the corresponding memory manipulation functions.

We augmented type-based static analysis with a data-flow analysis that handles most practical cases of unsafe pointer casts and casts between pointers and integers. If a value v is ever cast to a sensitive pointer type within the function being analyzed, or is passed as an argument or returned to another function where it is cast to a sensitive pointer, the analysis considers v to be sensitive as well. This analysis may fail when the data flow between v and its cast to a sensitive pointer type cannot be fully recovered statically, which might cause false violation reports (we have not observed any during our evaluation). Such casts are a common problem for all pointer-based memory safety mechanisms for C/C++ that do not require source code modifications [34].

A key benefit of CPI is its selectivity: the number of pointer operations deemed to be sensitive is a small fraction of all pointer operations in a program. As we show in §5, for SPEC CPU2006, the CPI type-based analysis identifies for instrumentation 6.5% of all pointer accesses; this translates into a reduction of performance overhead of 16 – 44× relative to full memory safety.

Nevertheless, we still think CPI can benefit from more sophisticated analyses. CPI can leverage any kind of *points-to* static analysis, as long as it provides an over-approximate set of sensitive pointers. For instance, when extending CPI to also protect select non-code-pointer data, we think DSA [27, 28] could prove more effective.

3.2.2 CPI Instrumentation

CPI instruments a program in order to (i) ensure that all sensitive pointers are stored in a safe region, (ii) create and propagate metadata for such pointers at runtime, and (iii) check the metadata on dereferences of such pointers.

In terms of memory layout, CPI introduces a safe region in addition to the regular memory region (Fig. 2). Storage space for sensitive pointers is allocated in both the safe region (the *safe pointer store*) and the regular region (as usual); one of the two copies always remains unused. This is necessary for universal pointers (e.g., void*), which could be stored in either region depending on whether they are sensitive at run time or not, and also helps to avoid some compatibility issues that arise from the change in memory layout. The address in regular memory is used as an offset to to look up the value of a sensitive pointer in the safe pointer store.

The *safe pointer store* maps the address &p of sensitive pointer p, as allocated in the regular region, to the value of p and associated metadata. The metadata for p describes the target object on which p is based: lower and upper address bounds of the object, and a temporal id (see Fig. 2). The layout of the safe pointer store is similar to metadata storage in SoftBounds+CETS [35], except that CPI *also* stores the value of p in the safe pointer store. Combined with the isolation of the safe region (§3.2.3), this allows CPI to guarantee full memory safety of all sensitive pointers without having to instru-

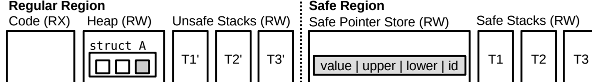


Figure 2: CPI memory layout: The safe region contains the safe pointer store and the safe stacks. The location of a sensitive pointer on the left (shaded) remains unused, while the value of this pointer and its metadata are stored in the safe pointer store. The safe stacks T_1, T_2, T_3 have corresponding stacks T_1', T_2', T_3' in regular memory to allocate unsafe stack objects.

ment all pointer operations.

The instrumentation step changes instructions that operate on sensitive pointers, as found by CPI's static analysis, to create and propagate the metadata directly following the based-on definition in §3.1. Instructions that explicitly take addresses of a statically allocated memory object or a function, allocate a new object on the heap, or take an address of a sub-object are instrumented to create metadata that describe the corresponding object. Instructions that compute pointer expressions are instrumented to propagate the metadata accordingly. Instructions that load or store sensitive pointers to memory are replaced with CPI intrinsic instructions (§3.2.3) that load or store both the pointer values and their metadata from/to the safe pointer store. In principle, call and return instructions also store and load code pointers, and so would need to be instrumented, but we instead protect return addresses using a safe stack (§3.2.4).

Every dereference of a sensitive pointer is instrumented to check at runtime whether it is safe, using the metadata associated with the pointer being dereferenced. Together with the restricted access to the safe region, this results in precise memory safety for all sensitive pointers.

Universal pointers (`void*` and `char*`) are stored in either the safe pointer store or the regular region, depending on whether they are sensitive at runtime or not. CPI instruments instructions that cast from non-sensitive to universal pointer types to assign special "invalid" metadata (e.g., with lower bound greater than the upper bound) for the resulting universal pointers. These pointers, as a result, would never be allowed to access the safe region. CPI intrinsics for universal pointers would only store a pointer in the safe pointer store if it had valid metadata, and only load it from the safe pointer store if it contained valid metadata for that pointer; otherwise, they would store/load from the regular region.

CPI can be configured to simultaneously store protected pointers in both the safe pointer store and regular regions, and check whether they match when loading them. In this debug mode, CPI detects all *attempts* to hijack control flow using non-protected pointer errors; in the default mode, such attempts are silently prevented. This debug mode also provides better compatibility with non-instrumented code that may read protected pointers (for example, callback addresses) but not write them.

Modern compilers contain powerful static analysis passes that can often prove statically that certain memory accesses are always safe. The CPI instrumentation pass precedes compiler optimizations, thus allowing them to potentially optimize away some of the inserted checks while preserving the security guarantees.

3.2.3 Isolating the Safe Region

The safe region can only be accessed via CPI intrinsic instructions, and they properly handle pointer metadata and the safe stack (§3.2.4). The mechanism for achieving this isolation is architecture-dependent.

On x86-32, we rely on hardware segment protection. We make the safe region accessible through a dedicated segment register, which is otherwise unused, and configure limits for all other segment registers to make the region inaccessible through them. The CPI intrinsics are then turned into code that uses the dedicated register and ensures that no other instructions in the program use that register. The segment registers are configured by the program loader, whose integrity we assume in our threat model; we also prevent the program from reconfiguring the segment registers via system calls. None of the programs we evaluated use the segment registers.

On x86-64, CPI relies on the fact that no addresses pointing into the safe region are ever stored in the regular region. This architecture no longer enforces the segment limits, however it still provides two segment registers with configurable base addresses. Similarly to x86-32, we use one of these registers to point to the safe region, however, we choose the base address of the safe region at random and rely on preventing access to it through information hiding. Unlike classic ASLR though, our hiding is leak-proof: since the objects in the safe region are indexed by addresses allocated for them in the regular region, no addresses pointing into the safe region are ever stored in regular memory at any time during execution. The 48-bit address space of modern x86-64 CPUs makes guessing the safe region address impractical, because most failed guessing attempts would crash the program, and such frequent crashes can easily be detected by other means.

Other architectures could use randomization-based protection as well, or rely on precise software fault isolation (SFI) [11]. SFI requires that all memory operations in a program are instrumented, but the instrumentation is lightweight: it could be as small as a single `and` operation if the safe region occupies the entire upper half of the address space of a process. In our experiments, the additional overhead introduced by SFI was less than 5%.

Since sensitive pointers form a small fraction of all data stored in memory, the safe pointer store is highly sparse. To save memory, it can be organized as a hash ta-

ble, a multi-level lookup table, or as a simple array relying on the sparse address space support of the underlying OS. We implemented and evaluated all three versions, and we discuss the fastest choice in §4.

In the future, we plan to leverage Intel MPX [24] for implementing the safe region, as described in §4.

3.2.4 The Safe Stack

CPI treats the stack specially, in order to reduce performance overhead and complexity. This is primarily because the stack hosts values that are accessed frequently, such as return addresses that are code pointers accessed on every function call, as well as spilled registers (temporary values that do not fit in registers and compilers store on the stack). Furthermore, tracking which of these values will end up at run time in memory (and thus need to be protected) vs. in registers is difficult, as the compiler decides which registers to spill only during late stages of code generation, long after CPI's instrumentation pass.

A key observation is that the safety of most accesses to stack objects can be checked statically during compilation, hence such accesses require no runtime checks or metadata. Most stack frames contain only memory objects that are accessed exclusively within the corresponding function and only through the stack pointer register with a constant offset. We therefore place all such proven-safe objects onto a *safe stack* located in the safe region. The safe stack can be accessed without any checks. For functions that have memory objects on their stack that do require checks (e.g., arrays or objects whose address is passed to other functions), we allocate separate stack frames in the regular memory region. In our experience, less than 25% of functions need such additional stack frames (see Table 2). Furthermore, this fraction is much smaller among short functions, for which the overhead of setting up the extra stack frame is non-negligible.

The safe stack mechanism consists of a static analysis pass, an instrumentation pass, and runtime support. The analysis pass identifies, for every function, which objects in its stack frame are guaranteed to be accessed safely and can thus be placed on the safe stack; return addresses and spilled registers always satisfy this criterion. For the objects that do not satisfy this criterion, the instrumentation pass inserts code that allocates a stack frame for these objects on the regular stack. The runtime support allocates regular stacks for each thread and can be implemented either as part of the threading library, as we did on FreeBSD, or by intercepting thread create/destroy, as we did on Linux. CPI stores the regular stack pointer inside the thread control block, which is pointed to by one of the segment registers and can thus be accessed with a single memory read or write.

Our safe stack layout is similar to double stack approaches in ASR [6] and XFI [18], which maintain a separate stack for arrays and variables whose addresses are taken. However, we use the safe stack to enforce the CPI property instead of implementing software fault isolation. The safe stack is also comparable to language-based approaches like Cyclone [25] or CCured [36] that simply allocate these objects on the heap, but our approach has significantly lower performance overhead.

Compared to a shadow stack like in CFI [1], which duplicates return instruction pointers outside of the attacker's access, the CPI safe stack presents several advantages: (i) all return instruction pointers and most local variables are protected, whereas a shadow stack only protects return instruction pointers; (ii) the safe stack is compatible with uninstrumented code that uses just the regular stack, and it directly supports exceptions, tail calls, and signal handlers; (iii) the safe stack has near-zero performance overhead (§5.2), because only a handful of functions require extra stack frames, while a shadow stack allocates a shadow frame for every function call.

The safe stack can be employed independently from CPI, and we believe it can replace stack cookies [14] in modern compilers. By providing precise protection of all return addresses (which are the target of ROP attacks today), spilled registers, and some local variables, the safe stack provides substantially stronger security than stack cookies, while incurring equal or lower performance overhead and deployment complexity.

3.3 Code-Pointer Separation (CPS)

The code-pointer separation property trades some of CPI's security guarantees for reduced runtime overhead. This is particularly relevant to C++ programs with many virtual functions, where the fraction of sensitive pointers instrumented by CPI can become high, since every pointer to an object that contains virtual functions is sensitive. We found that, on average, CPS reduces overhead by 4.3× (from 8.4% for CPI down to 1.9% for CPS), and in some cases by as much as an order of magnitude.

CPS further restricts the set of protected pointers to code pointers only, leaving pointers that point to code pointers uninstrumented. We additionally restrict the definition of based-on by requiring that a code pointer be based only on a control flow destination. This restriction prevents attackers from "forging" a code pointer from a value of another type, but still allows them to trick the program into reading or updating wrong code pointers.

CPS is enforced similarly to CPI, except (i) for the criteria used to identify sensitive pointers during static analysis, and (ii) that CPS does not need any metadata. Control-flow destinations (pointed to by code pointers) do not have bounds, because the pointer value must always match the destination exactly, hence no need for bounds metadata. Furthermore, they are typically static, hence do not need temporal metadata either (there are

a few rare exceptions, like unloading a shared library, which are handled separately). This reduces the size of the safe region and the number of memory accesses when loading or storing code pointers. If the safe region is organized as a simple array, a CPS-instrumented program performs essentially the same number of memory accesses when loading or storing code pointers as a non-instrumented one; the only difference is that the pointers are being loaded or stored from the safe pointer store instead of their original location (universal pointer load or store instructions still introduce one extra memory access per such instruction). As a result, CPS can be enforced with low performance overhead.

CPS guarantees that (i) code pointers can only be stored to or modified in memory by code pointer store instructions, and (ii) code pointers can only be loaded by code pointer load instructions from memory locations to which previously a code pointer store instruction stored a value. Combined with the safe stack, CPS precisely protects return addresses. CPS is stronger than most CFI implementations [1, 54, 53], which allow any vulnerable instruction in a program to modify any code pointer; they only check that the value of a code pointer (when used in an indirect control transfer) points to a function defined in a program (for function pointers) or directly follows a call instruction (for return addresses). CPS guarantee (i) above restricts the attack surface, while guarantee (ii) restricts the attacker's flexibility by limiting the set of locations to which the control can be redirected—the set includes only entry points of functions whose addresses were explicitly taken by the program.

To illustrate this difference, consider the case of the Perl interpreter, which implements its opcode dispatch by representing internally a Perl program as a sequence of function pointers to opcode handlers and then calling in its main execution loop these function pointers one by one. CFI statically approximates the set of legitimate control-flow targets, which in this case would include all possible Perl opcodes. CPS however permits only calls through function pointers that are actually assigned. This means that a memory bug in a CFI-protected Perl interpreter may permit an attacker to divert control flow and execute any Perl opcode, whereas in a CPS-protected Perl interpreter the attacker could at most execute an opcode that exists in the running Perl program.

CPS provides strong control-flow integrity guarantees and incurs low overhead (§5). We found that it prevents all recent attacks designed to bypass CFI [19, 15, 9]. We consider CPS to be a solid alternative to CPI in those cases when CPI's (already low) overhead seems too high.

4 Implementation

We implemented a CPI/CPS enforcement tool for C/C++, called Levee, on top of the LLVM 3.3 compiler infrastructure [30], with modifications to LLVM libraries, the clang compiler, and the compiler-rt runtime. To use Levee, one just needs to pass additional flags to the compiler to enable CPI (-fcpi), CPS (-fcps), or safe-stack protection (-fstack-protector-safe). Levee works on unmodified programs and supports Linux, FreeBSD, and Mac OS X in both 32-bit and 64-bit modes.

Levee can be downloaded from the project homepage `http://levee.epfl.ch`, and we plan to push our changes to the upstream LLVM.

Analysis and instrumentation passes: We implemented the static analysis and instrumentation for CPI as two LLVM passes, directly following the design from §3.2.1 and §3.2.2. The LLVM passes operate on the LLVM intermediate representation (IR), which is a low-level strongly-typed language-independent program representation tailored for static analyses and optimization purposes. The LLVM IR is generated from the C/C++ source code by clang, which preserves most of the type information that is required by our analysis, with a few corner cases. For example, in certain cases, clang does not preserve the original types of pointers that are cast to void* when passing them as an argument to memset or similar functions, which is required for the memset-related optimizations discussed in §3.2.2. The IR also does not distinguish between void* and char* (represents both as i8*), but this information is required for our string pointers detection heuristic. We augmented clang to always preserve such type information as LLVM metadata.

Safe stack instrumentation pass: The safe stack instrumentation targets functions that contain on-stack memory objects that cannot be put on the safe stack. For such functions, it allocates a stack frame on the unsafe stack and relocates corresponding variables to that frame.

Given that most of the functions do not need an unsafe stack, Levee uses the usual stack pointer (rsp register on x86-64) as the safe stack pointer, and stores the unsafe stack pointer in the thread control block, which is accessible directly through one of the segment registers. When needed, the unsafe stack pointer is loaded into an IR local value, and Levee relies on the LLVM register allocator to pick the register for the unsafe stack pointer. Levee explicitly encodes unsafe stack operations as IR instructions that manipulate an unsafe stack pointer; it leaves all operations that use a safe stack intact, letting the LLVM code generator manage them. Levee performs these changes as a last step before code generation (directly replacing LLVM's stack-cookie protection pass), thus ensuring that it operates on the final stack layout.

Certain low-level functions modify the stack pointer directly. These functions include setjmp/longjmp and exception handling functions (which store/load the stack pointer), and thread create/destroy functions, which allocate/free stacks for threads. On FreeBSD we provide

full-system CPI, so we directly modified these functions to support the dual stacks. On Linux, our instrumentation pass finds setjmp/longjmp and exception handling functions in the program and inserts required instrumentation at their call sites, while thread create/destroy functions are intercepted and handled by the Levee runtime.

Runtime support library: Most of the instrumentation by the above passes are added as intrinsic function calls, such as cpi_ptr_store() or cpi_memcpy(), which are implemented by Levee's runtime support library (a part of compiler-rt). This design cleanly separates the safe pointer store implementation from the instrumentation pass. In order to avoid the overhead associated with extra function calls, we ensure that some of the runtime support functions are always inlined. We compile these functions into LLVM bitcode and instruct clang to link this bitcode into every object file it compiles. Functions that are called rarely (e.g., cpi_abort(), called when a CPI violation is detected) are never inlined, in order to reduce the instruction cache footprint of the instrumentation.

We implemented and benchmarked several versions of the safe pointer store map in our runtime support library: a simple array, a two-level lookup table, and a hashtable. The array implementation relies on the sparse address space support of the underlying OS. Initially we found it to perform poorly on Linux, due to many page faults (especially at startup) and additional TLB pressure. Switching to superpages (2 MB on Linux) made this simple table the fastest implementation of the three.

Binary level functionality: Some code pointers in binaries are generated by the compiler and/or linker, and cannot be protected on the IR level. Such pointers include the ones in jump tables, exception handler tables, and the global offset table. Bounds checks for the jump tables and the exception handler tables are already generated by LLVM anyway, and the tables themselves are placed in read-only memory, hence cannot be overwritten. We rely on the standard loader's support for read-only global offset tables, using the existing RTLD_NOW flag.

Limitations: The CPI design described in §3 includes both spatial and temporal memory safety enforcement for sensitive pointers, however our current prototype implements spatial memory safety only. It can be easily extended to enforce temporal safety by directly applying the technique described in [35] for sensitive pointers.

Levee currently supports Linux, FreeBSD and Mac OS user-space applications. We believe Levee can be ported to protect OS kernels as well. Related technical challenges include integration with the kernel memory management subsystem and handling of inline assembly.

CPI and CPS require instrumenting all code that manipulates sensitive pointers; non-instrumented code can cause unnecessary aborts. Non-instrumented code could come from external libraries compiled without Levee, inline assembly, or dynamically generated code. Levee can be configured to simultaneously store sensitive pointers in both the safe and the regular regions, in which case non-instrumented code works fine as long as it only reads sensitive pointers but doesn't write them.

Inline assembly and dynamically generated code can still update sensitive pointers if instrumented with appropriate calls to the Levee runtime, either manually by a programmer or directly by the code generator.

Dynamically generated code (e.g., for JIT compilation) poses an additional problem: running the generated code requires making writable pages executable, which violates our threat model (this is a common problem for most control-flow integrity mechanisms). One solution is to use hardware or software isolation mechanisms to isolate the code generator from the code it generates.

Sensitive data protection: Even though the main focus of CPI is control-flow hijack protection, the same technique can be applied to protect other types of sensitive data. Levee can treat programmer-annotated data types as sensitive and protect them just like code pointers. CPI could also selectively protect individual program variables (as opposed to types), however it would require replacing the type-based static analysis described in §3.2.1 with data-based points-to analysis such as DSA [27, 28].

Future MPX-based implementation: Intel announced a hardware extension, Intel MPX, to be used for hardware-enforced memory safety [23]. It is proposed as a testing tool, probably due to the associated overhead; no overhead numbers are available at the time of writing.

We believe MPX (or similar) hardware can be repurposed to enforce CPI with lower performance overhead than our existing software-only implementation. MPX provides special registers to store bounds along with instructions to check them, and a hardware-based implementation of a pointer metadata store (analogous to the safe pointer store in our design), organized as a two-level lookup table. Our implementation can be adapted to use these facilities once MPX-enabled hardware becomes available. We believe that a hardware-based CPI implementation can reduce the overhead of a software-only CPI in much the same way as HardBound [16] or Watchdog [33] reduced the overhead of SoftBound.

Adopting MPX for CPI might require implementing metadata loading logic in software. Like CPI, MPX also stores the pointer value together with the metadata. However, being a testing tool, MPX chooses compatibility with non-instrumented code over security guarantees: it uses the stored pointer value to check whether the original pointer was modified by non-instrumented code and, if yes, resets the bounds to $[0, \infty]$. In contrast, CPI's guarantees depend on preventing any non-instrumented code from ever modifying sensitive pointer values.

5 Evaluation

In this section we evaluate Levee's effectiveness, efficiency, and practicality. We experimentally show that both CPI and CPS are 100% effective on RIPE, the most recent attack benchmark we are aware of (§5.1). We evaluate the efficiency of CPI, CPS, and the safe stack on SPEC CPU2006, and find average overheads of 8.4%, 1.9%, and 0% respectively (§5.2). To demonstrate practicality, we recompile with CPI/CPS/ safe stack the base FreeBSD plus over 100 packages and report results on several benchmarks (§5.3).

We ran our experiments on an Intel Xeon E5-2697 with 24 cores @ 2.7GHz in 64-bit mode with 512GB RAM. The SPEC benchmarks ran on an Ubuntu Precise Pangolin (12.04 LTS), and the FreeBSD benchmarks in a KVM-based VM on this same system.

5.1 Effectiveness on the RIPE Benchmark

We described in §3 the security guarantees provided by CPI, CPS, and the safe stack based on their design; to experimentally evaluate their effectiveness, we use the RIPE [49] benchmark. This is a program with many different security vulnerabilities and a set of 850 exploits that attempt to perform control-flow hijack attacks on the program using various techniques.

Levee deterministically prevents all attacks, both in CPS and CPI mode; when using only the safe stack, it prevents all stack-based attacks. On vanilla Ubuntu 6.06, which has no built-in defense mechanisms, 833–848 exploits succeed when Levee is not used (some succeed probabilistically, hence the range). On newer systems, fewer exploits succeed, due to built-in protection mechanisms, changes in the run-time layout, and compatibility issues with the RIPE benchmark. On vanilla Ubuntu 13.10, with all protections (DEP, ASLR, stack cookies) disabled, 197–205 exploits succeed. With all protections enabled, 43–49 succeed. With CPS or CPI, none do.

The RIPE benchmark only evaluates the effectiveness of preventing existing attacks; as we argued in §3 and according to the proof outlined in Appendix A, CPI renders all (known and unknown) memory corruption-based control-flow hijack attacks impossible.

5.2 Efficiency on SPEC CPU2006 Benchmarks

In this section we evaluate the runtime overhead of CPI, CPS, and the safe stack. We report numbers on all SPEC CPU2006 benchmarks written in C and C++ (our prototype does not handle Fortran). The results are summarized in Table 1 and presented in detail in Fig. 3. We also compare Levee to two related approaches, SoftBound [34] and control-flow integrity [1, 54, 53].

CPI performs well for most C benchmarks, however it can incur higher overhead for programs written in C++. This overhead is caused by abundant use of pointers to

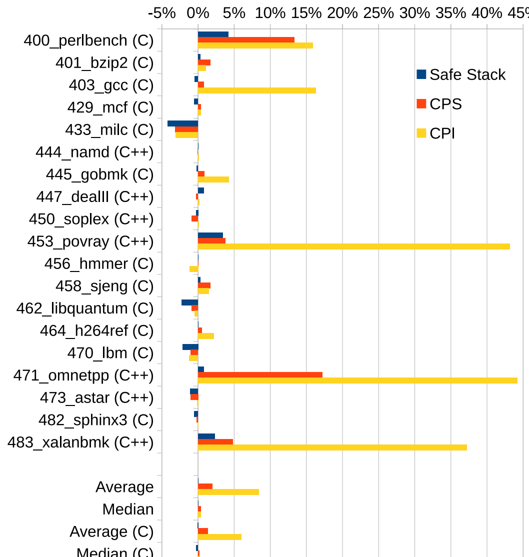


Figure 3: Levee performance for SPEC CPU2006, under three configurations: full CPI, CPS only, and safe stack only.

	Safe Stack	CPS	CPI
Average (C/C++)	0.0%	1.9%	8.4%
Median (C/C++)	0.0%	0.4%	0.4%
Maximum (C/C++)	4.1%	17.2%	44.2%
Average (C only)	-0.4%	1.2%	2.9%
Median (C only)	-0.3%	0.5%	0.7%
Maximum (C only)	4.1%	13.3%	16.3%

Table 1: Summary of SPEC CPU2006 performance overheads.

C++ objects that contain virtual function tables—such pointers are sensitive for CPI, and so all operations on them are instrumented. Same reason holds for gcc: it embeds function pointers in some of its data structures and then uses pointers to these structures frequently.

The next-most important source of overhead are libc memory manipulation functions, like memset and memcpy. When our static analysis cannot prove that a call to such a function uses as arguments only pointers to non-sensitive data, Levee replaces the call with one to a custom version of an equivalent function that checks the safe pointer store for each updated/copied word, which introduces overhead. We expect to remove some of this overhead using improved static analysis and heuristics.

CPS averages 1.2–1.8% overhead, and exceeds 5% on only two benchmarks, omnetpp and perlbench. The former is due to the large number of virtual function calls occurring at run time, while the latter is caused by a specific way in which perl implements its opcode dispatch: it internally represents a program as a sequence of function pointers to opcode handlers, and its main execution loop calls these function pointers one after the other. Most other interpreters use a switch for opcode dispatch.

Safe stack provided a surprise: in 9 cases (out of

19), it improves performance instead of hurting it; in one case (namd), the improvement is as high as 4.2%, more than the overhead incurred by CPI and CPS. This is because objects that end up being moved to the regular (unsafe) stack are usually large arrays or variables that are used through multiple stack frames. Moving such objects away from the safe stack increases the locality of frequently accessed values on the stack, such as CPU register values temporarily stored on the stack, return addresses, and small local variables.

The safe stack overhead exceeds 1% in only three cases, perlbench, xalanbmk, and povray. We studied the disassembly of the most frequently executed functions that use unsafe stack frames in these programs and found that some of the overhead is caused by inefficient handling of the unsafe stack pointer by LLVM's register allocator. Instead of keeping this pointer in a single register and using it as a base for all unsafe stack accesses, the program keeps moving the unsafe stack pointer between different registers and often spills it to the (safe) stack. We believe this can be resolved by making the register allocator algorithm aware of the unsafe stack pointer.

In contrast to the safe stack, stack cookies deployed today have an overhead of up to 5%, and offer strictly weaker protection than our safe stack implementation.

The data structures used for the safe stack and the safe memory region result in memory overhead compared to a program without protection. We measure the memory overhead when using either a simple array or a hash table. For SPEC CPU2006 the median memory overhead for the safe stack is 0.1%; for CPS the overhead is 2.1% for the hash table and 5.6% for the array; and for CPI the overhead is 13.9% for the hash table and 105% for the array. We did not optimize the memory overhead yet and believe it can be improved in future prototypes.

In Table 2 we show compilation statistics for Levee. The first column shows that only a small fraction of all functions require an unsafe stack frame, confirming our hypothesis from §3.2.4. The other two columns confirm the key premises behind our approach, namely that CPI requires much less instrumentation than full memory safety, and CPS needs much less instrumentation than CPI. The numbers also correlate with Fig. 3.

Benchmark	FN_{UStack}	MO_{CPS}	MO_{CPI}
400_perlbench	15.0%	1.0%	13.8%
401_bzip2	27.2%	1.3%	1.9%
403_gcc	19.9%	0.3%	6.0%
429_mcf	50.0%	0.5%	0.7%
433_milc	50.9%	0.1%	0.7%
444_namd	75.8%	0.6%	1.1%
445_gobmk	10.3%	0.1%	0.4%
447_dealII	12.3%	6.6%	13.3%
450_soplex	9.5%	4.0%	2.5%
453_povray	26.8%	0.8%	4.7%
456_hmmer	13.6%	0.2%	2.0%
458_sjeng	50.0%	0.1%	0.1%
462_libquantum	28.5%	0.4%	2.3%
464_h264ref	20.5%	1.5%	2.8%
470_lbm	16.6%	0.6%	1.5%
471_omnetpp	6.9%	10.5%	36.6%
473_astar	9.0%	0.1%	3.2%
482_sphinx3	19.7%	0.1%	4.6%
483_xalancbmk	17.5%	17.5%	27.1%

Table 2: Compilation statistics for Levee: FN_{UStack} lists what fraction of functions need an unsafe stack frame; MO_{CPS} and MO_{CPI} show the fraction of memory operations instrumented for CPS and CPI, respectively.

Benchmark	Safe Stack	CPS	CPI	SoftBound
401_bzip2	0.3%	1.2%	2.8%	90.2%
447_dealII	0.8%	-0.2%	3.7%	60.2%
458_sjeng	0.3%	1.8%	2.6%	79.0%
464_h264ref	0.9%	5.5%	5.8%	249.4%

Table 3: Overhead of Levee and SoftBound on SPEC programs that compile and run errors-free with SoftBound.

Comparison to SoftBound: We compare with SoftBound [34] on the SPEC benchmarks. We cannot fairly reuse the numbers from [34], because they are based on an older version of SPEC. In Table 3 we report numbers for the four C/C++ SPEC benchmarks that can compile with the current version of SoftBound. This comparison confirms our hypothesis that CPI requires significantly lower overhead compared to full memory safety.

Theoretically, CPI suffers from the same compatibility issues (e.g., handling unsafe pointer casts) as pointer-based memory safety. In practice, such issues arise much less frequently for CPI, because CPI instruments much fewer pointers. Many of the SPEC benchmarks either don't compile or terminate with an error when instrumeted by SoftBound, which illustrates the practical impact of this difference.

Comparison to control-flow integrity (CFI): The average overhead for compiler-enforced CFI is 21% for a subset of the SPEC CPU2000 benchmarks [1] and 5-6% for MCFI [39] (without stack pointer integrity). CCFIR [53] reports an overhead of 3.6%, and binCFI [54] reports 8.54% for SPEC CPU2006 to enforce a weak CFI property with globally merged target sets. WIT [3], a source-based mechanism that enforces both CFI and write integrity protection, has 10% overhead[1].

At less than 2%, CPS has the lowest overhead among all existing CFI solutions, while providing stronger protection guarantees. Also, CPI's overhead is bested only by CCFIR. However, unlike any CFI mechanism, CPI guarantees the impossibility of any control-flow hijack attack based on memory corruptions. In contrast, there

[1] We were unable to find open-source implementations of compiler-based CFI, so we can only compare to published overhead numbers.

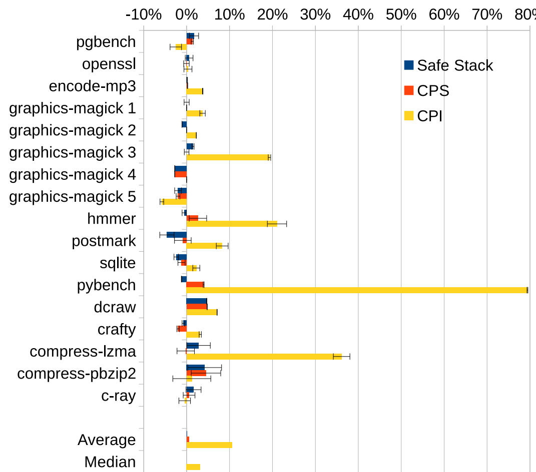


Figure 4: Performance overheads on FreeBSD (Phoronix).

exist successful attacks against CFI [19, 15, 9]. While neither of these attacks are possible against CPI by construction, we found that, in practice, neither of them would work against CPS either. We further discuss conceptual differences between CFI and CPI in §6.

5.3 Case Study: A Safe FreeBSD Distribution

Having shown that Levee is both effective and efficient, we now evaluate the feasibility of using Levee to protect an entire operating system distribution, namely FreeBSD 10. We rebuilt the base system—base libraries, development tools, and services like bind and openssh—plus more than 100 packages (including apache, postgresql, php, python) in four configurations: CPI, CPS, Safe Stack, and vanilla. FreeBSD 10 uses LLVM/clang as its default compiler, while some core components of Linux (e.g., glibc) cannot be built with clang yet. We integrated the CPI runtime directly into the C library and the threading library. We have not yet ported the runtime to kernel space, so the OS kernel remained uninstrumented.

We evaluated the performance of the system using the Phoronix test suite [41], a widely used comprehensive benchmarking platform for operating systems. We chose the "server" setting and excluded benchmarks marked as unsupported or that do not compile or run on recent FreeBSD versions. All benchmarks that compiled and worked on vanilla FreeBSD also compiled and worked in the CPI, CPS and Safe Stack versions.

Fig. 4 shows the overhead of CPI, CPS and the safe-stack versions compared to the vanilla version. The results are consistent with the SPEC results presented in §5.2. The Phoronix benchmarks exercise large parts of the system and some of them are multi-threaded, which introduces significant variance in the results, especially when run on modern hardware. As Fig. 4 shows, for many benchmarks the overheads of CPS and the safe stack are within the measurement error.

Benchmark	Safe Stack	CPS	CPI
Static page	1.7%	8.9%	16.9%
Wsgi test page	1.0%	4.0%	15.3%
Dynamic page	1.4%	15.9%	138.8%

Table 4: Throughput benchmark for web server stack (FreeBSD + Apache + SQLite + mod_wsgi + Python +Django).

We also evaluated a realistic usage model of the FreeBSD system as a web server. We installed Mezzanine, a content management system based on Django, which uses Python, SQLite, Apache, and mod_wsgi. We used the Apache ab tool to benchmark the throughput of the web server. The results are summarized in Table 4.

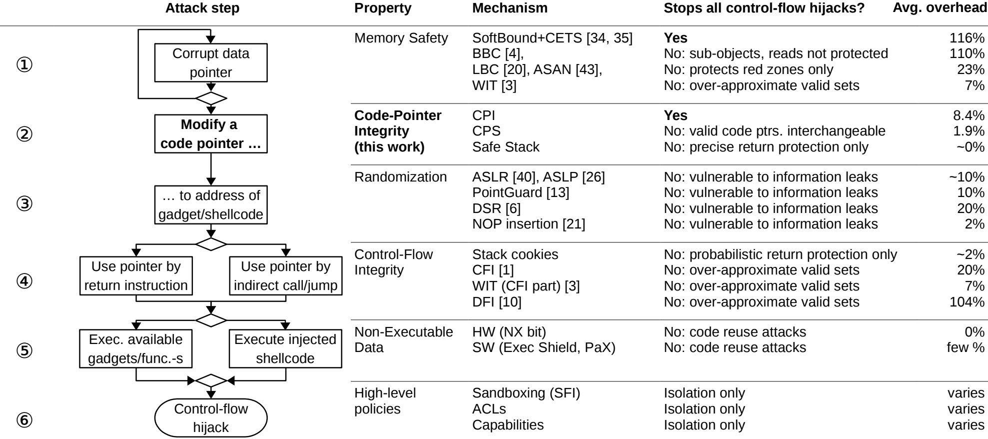
The CPI overhead for a dynamic page generated by Python code is much larger then we expected, but consistent with suspiciously high overhead of the pybench benchmark in Fig. 4. We think it might be caused by the use of some C constructs in the Python interpreter that are not yet handled well by our optimization heuristics, e.g., emulating C++ inheritance in C. We believe the performance might be improved in this case by extending the heuristics to recognize such C constructs.

6 Related Work

A variety of defense mechanisms have been proposed to-date to answer the increasing challenge of control-flow hijack attacks. Fig. 5 compares the design of the different protection approaches to our approach.

Enforcing memory safety ensures that no dangling or out-of-bounds pointers can be read or written by the application, thus preventing the attack in its first step. Cyclone [25] and CCured [36] extend C with a safe type system to enforce memory safety features. These approaches face the problem that there is a large (unported) legacy code base. In contrast, CPI and CPS both work for unmodified C/C++ code. SoftBound [34] with its CETS [35] extension enforces *complete* memory safety at the cost of 2× – 4× slowdown. Tools with less overhead, like BBC [4], only *approximate* memory safety. LBC [20] and Address Sanitizer [43] detect continuous buffer overflows and (probabilistically) indexing errors, but can be bypassed by an attacker who avoids the red zones placed around objects. Write integrity testing (WIT) [3] provides spatial memory safety by restricting pointer writes according to points-to sets obtained by an over-approximate static analysis (and is therefore limited by the static analysis). Other techniques [17, 2] enforce type-safe memory reuse to mitigate attacks that exploit temporal errors (use after frees).

CPI by design enforces spatial and temporal memory safety for a subset of data (code pointers) in Step 2 of Fig. 5. Our Levee prototype currently enforces spatial memory safety and may be extended to enforce temporal memory safety as well (e.g., how CETS extends Soft-



Property	Mechanism	Stops all control-flow hijacks?	Avg. overhead
Memory Safety	SoftBound+CETS [34, 35]	**Yes**	116%
	BBC [4],	No: sub-objects, reads not protected	110%
	LBC [20], ASAN [43],	No: protects red zones only	23%
	WIT [3]	No: over-approximate valid sets	7%
Code-Pointer Integrity (this work)	CPI	**Yes**	8.4%
	CPS	No: valid code ptrs. interchangeable	1.9%
	Safe Stack	No: precise return protection only	~0%
Randomization	ASLR [40], ASLP [26]	No: vulnerable to information leaks	~10%
	PointGuard [13]	No: vulnerable to information leaks	10%
	DSR [6]	No: vulnerable to information leaks	20%
	NOP insertion [21]	No: vulnerable to information leaks	2%
Control-Flow Integrity	Stack cookies	No: probabilistic return protection only	~2%
	CFI [1]	No: over-approximate valid sets	20%
	WIT (CFI part) [3]	No: over-approximate valid sets	7%
	DFI [10]	No: over-approximate valid sets	104%
Non-Executable Data	HW (NX bit)	No: code reuse attacks	0%
	SW (Exec Shield, PaX)	No: code reuse attacks	few %
High-level policies	Sandboxing (SFI)	Isolation only	varies
	ACLs	Isolation only	varies
	Capabilities	Isolation only	varies

Figure 5: Summary of control-flow hijack defense mechanisms aligned with individual steps that are necessary for a successful attack. The figure on the left is a simplified version of the complete memory corruption diagram in [46].

Bound). We believe CPI is the first to stop all control-flow hijack attacks at this step.

Randomization techniques, like ASLR [40] and ASLP [26], mitigate attacks by restricting the attacker's knowledge of the memory layout of the application in Step 3. PointGuard [13] and DSR [7] (which is similar to probabilistic WIT) randomize the data representation by encrypting pointer values, but face compatibility problems. Software diversity [21] allows fine-grained, per-instance code randomization. Randomization techniques are defeated by information leaks through, e.g., memory corruption bugs [45] or side channel attacks [22].

Control-flow integrity [1] ensures that the targets of all indirect control-flow transfers point to valid code locations in Step 4. All CFI solutions rely on statically pre-computed context-insensitive sets of valid control-flow target locations. Many practical CFI solutions simply include every function in a program in the set of valid targets [53, 54, 29, 47]. Even if precise static analysis was be feasible, CFI could not guarantee protection against all control-flow hijack attacks, but rather merely restrict the sets of potential hijack targets. Indeed, recent results [19, 15, 9] show that many existing CFI solutions can be bypassed in a principled way. CFI+SFI [52], Strato [51] and MIPS [38] enforce an even more relaxed, statically defined CFI property in order to enforce software-based fault isolation (SFI). CCFI [31] encrypts code pointers in memory and provides security guarantees close to CPS. Data-flow based techniques like data-flow integrity (DFI) [10] or dynamic taint analysis (DTA) [42] can enforce that the used code pointer was not set by an unrelated instruction or to untrusted data, respectively. These techniques may miss some attacks or cause false positives, and have higher performance costs than CPI and CPS. Stack cookies, CFI, DFI, and DTA protect control-transfer instructions by detecting illegal modification of the code pointer whenever it is used, while CPI protects the load and store of a code pointer, thus preventing the corruption in the first place. CPI provides precise and provable security guarantees.

In Step 5, the execution of injected code is prevented by enforcing the non-executable (NX) data policy, but code-reuse attacks remain possible.

High level policies, e.g., restricting the allowed system calls of an application, limit the power of the attacker even in the presence of a successful control-flow hijack attack in Step 6. Software fault isolation (SFI) techniques [32, 18, 11, 50, 52] restrict indirect control-flow transfers and memory accesses to part of the address space, enforcing a sandbox that contains the attack. SFI prevents an attack from escaping the sandbox and allows the enforcement of a high-level policy, while CPI enforces the control-flow inside the application.

7 Conclusion

This paper describes *code-pointer integrity* (CPI), a way to protect systems against all control-flow hijacks that exploit memory bugs, and *code-pointer separation*, a relaxed form of CPI that still provides strong guarantees. The key idea is to selectively provide full memory safety for just a subset of a program's pointers, namely code pointers. We implemented our approach and showed that it is effective, efficient, and practical. Given its advantageous security-to-overhead ratio, we believe our approach marks a step toward deterministically secure systems that are fully immune to control-flow hijack attacks.

Acknowledgments

We thank the anonymous reviewers and our shepherd Junfeng Yang for their valuable input. We are grateful to Martin Abadi, Herbert Bos, Miguel Castro, Vijay D'Silva, Ulfar Erlingsson, Johannes Kinder, Per Larsen, Jim Larus, Santosh Nagarakatte, and Jonas Wagner for

Atomic Types	a	$::=$ `int` $\mid p*$
Pointer Types	p	$::= a \mid s \mid f \mid$ `void`
Struct Types	s	$::=$ `struct`$\{\ldots; a_i : id_i; \ldots\}$
LHS Expressions	lhs	$::= x \mid *lhs \mid lhs.id \mid lhs\text{->}id$
RHS Expressions	rhs	$::= i \mid \&f \mid rhs + rhs \mid lhs \mid \&lhs$
		$\mid (a)\ rhs \mid$ `sizeof(p)` $\mid$ `malloc`(rhs)
Commands	c	$::= c;c \mid lhs = rhs \mid f() \mid (*lhs)()$

Figure 6: The subset of C used in Appendix A; x denotes local statically typed variables, id – structure fields, i – integers, and f – functions from a pre-defined set.

their valuable feedback and discussions on earlier versions of the paper. This work was supported by ERC Starting Grant No. 278656, a Microsoft Research PhD fellowship, a gift from Google, DARPA award HR0011-12-2-005, NSF grants CNS-0831298 and CNS-1319137, and AFOSR FA9550-09-1-0539.

A Formal Model of CPI

This section presents a formal model and operational semantics of the CPI property and a sketch of its correctness proof. Due to the size and complexity of C/C++ specifications, we focus on a small subset of C that illustrates the most important features of CPI. Due to space limitations we focus on spatial memory safety. We build upon the formalization of spatial memory safety in SoftBound [34], reuse the same notation, and extend it to support applying spatial memory safety to a subset of memory locations. The formalism can be easily extended to provide temporal memory safety, directly applying the CETS [35] mechanism to the safe memory region of the model. Fig. 6 gives the syntax rules of the C subset we consider in this section. All valid programs must also pass type checking as specified by the C standard.

We define the runtime environment E of a program as a triple (S, M_u, M_s), where S maps variable identifiers to their respective atomic types and addresses, a regular memory M_u maps addresses to values (denoted as v and called regular values), and a safe memory M_s maps addresses to values with bounds information (denoted as $v_{(b,e)}$ and called safe values) or a special marker `none`. The bounds information specifies the lowest (b) and the highest (e) address of the corresponding memory object. M_u and M_s use the same addressing, but might contain distinct values for the same address. Some locations (e.g., of `void*` type) can store either safe or regular value and are resolved to either M_s or M_u at runtime.

The runtime provides the usual set of memory operations for M_u and M_s, as summarized in Table 5. M_u models standard memory, whereas M_s stores values with bounds and has a special marker for “absent” locations, similarly to the memory in SoftBound’s [34] formalization. We assume the memory operations follow the standard behavior of read/write/malloc operations in all other

Operation	Semantics
$\texttt{read}_u\ M_u\ l$	return $M_u[l]$
$\texttt{write}_u\ M_u\ l\ v$	set $M_u[l] = v$
$\texttt{read}_s\ M_s\ l$	return $M_s[l]$, if l is allocated; return `none` otherwise
$\texttt{write}_s\ M_s\ l\ v_{(b,e)}$	set $M_s[l] = v_{(b,e)}$, if l is allocated; do nothing otherwise
$\texttt{write}_s\ M_s\ l$ `none`	set $M_s[l] =$ `none`, if l is allocated; do nothing otherwise
`malloc` $E\ i$	allocate a memory object of size i in both $E.M_u$ and $E.M_s$ (at the same address); fail when out of memory

Table 5: Memory Operations in CPI

`sensitive`	`int`	$::= \mathit{false}$
`sensitive`	`void`	$::= \mathit{true}$
`sensitive`	f	$::= \mathit{true}$
`sensitive`	$p*$	$::=$ `sensitive` p
`sensitive`	s	$::= \bigvee_{i \in \mathit{fields\ of\ s}}$ `sensitive` a_i

Figure 7: The decision criterion for protecting types in CPI

respects, e.g., read returns the value previously written to the same location, malloc allocates a region of memory that is disjoint with any other allocated region, etc..

Enforcing the CPI property with low performance overhead requires placing most variables in M_u, while still ensuring that all pointers that require protection at runtime according to the CPI property are placed in M_s. In this formalization, we rely on type-based static analysis as defined by the `sensitive` criterion, shown on Fig. 7. We say a type p is sensitive iff `sensitive` $p = \mathit{true}$. Setting `sensitive` to true for all types would make the CPI operational semantics equivalent to the one provided by SoftBound and would ensure full spatial memory safety of all memory operations in a program.

The classification provided by the `sensitive` criterion is static and only determines which operations in a program to instrument. Expressions of sensitive types could evaluate to both safe or regular values at runtime, whereas expressions of regular types always evaluate to regular values. In particular, according to Fig. 7, `void*` is sensitive and, hence, in agreement with the C specification, values of that type can hold any pointer value at runtime, either safe or regular.

We extend the SoftBound definition of the result of an operation to differentiate between safe and regular values and left-hand-side locations:

$$\text{Results} \quad r ::= v_{(b,e)} \mid v \mid l_s \mid l_u \mid \texttt{OK} \mid \texttt{OutOfMem} \mid \texttt{Abort}$$

where $v_{(b,e)}$ and v are the safe (with bounds information) and, respectively, regular values that result from a right hand side expression, l_u and l_s are locations that result from a safe and regular left-hand-side expression, `OK` is a result of a successful command, and `OutOfMem` and `Abort` are error codes. We assume that all operational semantics rules of the language propagate these error codes up to the end of the program unchanged.

Using the above definitions, we now formalize the op-

erational semantics of CPI through three classes of rules. The $(E, lhs) \Rightarrow_l l_s : a$ and $(E, lhs) \Rightarrow_l l_u : a$ rules specify how left hand side expressions are evaluated to a safe or regular locations, respectively. The $(E, rhs) \Rightarrow_r (v_{(b,e)}, E')$ and $(E, rhs) \Rightarrow_r (v, E')$ rules specify how right hand side expressions are evaluated to safe values with bounds or regular values, respectively, possibly modifying the environment through memory allocation (turning it from E to E'). Finally, the $(E, c) \Rightarrow_c (r, E')$ rules specify how commands are executed, possibly modifying the environment, where r can be either OK or an error code. We only present the rules that are most important for the CPI semantics, omitting rules that simply represent the standard semantics of the C language.

Bounds information is initially assigned when allocating a memory object or when taking a function's address (both operations always return safe values):

$$\frac{\texttt{address}(f) = l}{(E, \&f) \Rightarrow_r (l_{(l,l)})} \qquad \frac{(E, rhs) = i \quad \texttt{malloc}\,E\,i = (l, E')}{(E, \texttt{malloc}(i)) \Rightarrow_r (l_{(l,l+i)}, E')}$$

Taking the address of a variable from S if its type is sensitive is analogous. Structure field access operations either narrow bounds information accordingly, or strip it if the type of the accessed field is regular.

Type casting results in a safe value iff a safe value is cast to a sensitive type:

$$\frac{\texttt{sensitive}\,a' \quad (E, rhs) \Rightarrow_l v_{(b,e)} : a}{(E, (a')rhs) \Rightarrow_r (v_{(b,e)}, E)} \qquad \frac{\neg\texttt{sensitive}\,a' \quad (E, rhs) \Rightarrow_l v_{(b,e)} : a}{(E, (a')rhs) \Rightarrow_r (v, E)}$$

$$\frac{(E, rhs) \Rightarrow_l v : a}{(E, (a')rhs) \Rightarrow_r (v, E)}$$

The next set of rules describes memory operations (pointer dereference and assignment) on sensitive types and safe values:

$$\frac{\texttt{sensitive}\,a \quad (E, lhs) \Rightarrow_l l_s : a* \quad \texttt{read}_s(E.M_s)l_s = \texttt{some}\,l'_{(b,e)} \quad l' \in [b, e - \texttt{sizeof}(a)]}{(E, *lhs) \Rightarrow_l l'_s : a}$$

$$\frac{\texttt{sensitive}\,a \quad (E, lhs) \Rightarrow_l l_s : a* \quad \texttt{read}_s(E.M_s)l_s = \texttt{some}\,l'_{(b,e)} \quad l' \notin [b, e - \texttt{sizeof}(a)]}{(E, *lhs) \Rightarrow_l \texttt{Abort}}$$

$$\frac{\texttt{sensitive}\,a \quad (E, lhs) \Rightarrow_l l_s : a \quad (E, rhs) \Rightarrow_r v_{(b,e)} : a \quad E'.M_s = \texttt{write}_\texttt{s}(E.M_s)l_s\,v_{(b,e)}}{(E, lhs = rhs) \Rightarrow_c (\texttt{OK}, E')}$$

These rules are identical to the corresponding rules of SoftBound [34] and ensure full spatial memory safety of all memory objects in the safe memory. Only operations matching those rules are allowed to access safe memory M_s. In particular, any attempts to access values of sensitive types through regular lvalues cause aborts:

$$\frac{\texttt{sensitive}\,a \quad (E, lhs) \Rightarrow_l l_u : a*}{(E, *lhs) \Rightarrow_l \texttt{Abort}} \qquad \frac{\texttt{sensitive}\,a \quad (E, lhs) \Rightarrow_l l_u : a}{(E, lhs = rhs) \Rightarrow_c (\texttt{Abort}, E)}$$

Note that these rules can only be invoked if the value of the sensitive type was obtained by casting from a regular type using a corresponding type casting rule. Levee relaxes the casting rules to allow propagation of bounds information through certain right-hand-side expressions of regular types. This relaxation handles most common cases of unsafe type casting; it affects performance (inducing more instrumentation) but not correctness.

Some sensitive types (only `void*` in our simplified version of C), can hold regular values at runtime. For example, a variable of `void*` type can first be used to store a function pointer and subsequently re-used to store an `int*` value. The following rules handle such cases:

$$\frac{\texttt{sensitive}\,a \quad (E, lhs) \Rightarrow_l l_s : a* \quad \texttt{read}_s(E.M_s)l = \texttt{none} \quad \texttt{read}_u(E.M_u)l = l'}{(E, *lhs) \Rightarrow_l l'_u : a}$$

$$\frac{\texttt{sensitive}\,a \quad (E, lhs) \Rightarrow_l l_s : a \quad (E, rhs) \Rightarrow_r v : a \quad E'.M_u = \texttt{write}_\texttt{u}(E.M_u)\,l\,v \quad E'.M_s = \texttt{write}_\texttt{s}(E.M_s)\,l\,\texttt{none}}{(E, lhs = rhs) \Rightarrow_c (\texttt{OK}, E')}$$

Memory operations on regular types always access regular memory, without any additional runtime checks, following the unsafe memory semantics of C.

$$\frac{\neg\texttt{sensitive}\,a \quad (E, lhs) \Rightarrow_l l : a* \quad \texttt{read}_u(E.M_u)l = l'}{(E, *lhs) \Rightarrow_l l'_u : a} \qquad \frac{\neg\texttt{sensitive}\,a \quad (E, lhs) \Rightarrow_l l : a \quad (E, rhs) \Rightarrow_r v : a \quad E'.M_u = \texttt{write}_\texttt{u}(E.M_u)\,l\,v}{(E, lhs = rhs) \Rightarrow_c (\texttt{OK}, E')}$$

These accesses to regular memory can go out of bounds but, given that $\texttt{read}_u$ and $\texttt{write}_u$ operations can only modify regular memory M_u, it does not violate memory safety of the safe memory.

Finally, indirect calls abort if the function pointer being called is not safe:

$$\frac{(E, lhs) \Rightarrow_r l_s : f*}{(E, (*lhs)()) \Rightarrow_c (\texttt{OK}, E')} \qquad \frac{(E, lhs) \Rightarrow_r l_u : f*}{(E, (*lhs)()) \Rightarrow_c (\texttt{Abort}, E)}$$

Note that the operational rules for values that are safe at runtime are fully equivalent to the corresponding SoftBound rules [34] and, therefore, satisfy the SoftBound safety invariant which, as proven in [34], ensures memory safety for these values. According to the `sensitive` criterion and the safe location dereference and indirect function call rules above, all dereferences of pointers that require protection according to the CPI property are always safe at runtime, or the program aborts. Therefore, the operational semantics defined above indeed ensure the CPI property as defined in §3.1.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conf. on Computer and Communication Security*, 2005.

[2] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, 2010.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symp. on Security and Privacy*, May 2008.

[4] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors. In *USENIX Security Symposium*, 2009.

[5] G. Altekar and I. Stoica. Focus replay debugging effort on the control plane. *USENIX Workshop on Hot Topics in Dependability*, 2010.

[6] S. Bhatkar, E. Bhatkar, R. Sekar, and D. C. Duvarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[7] S. Bhatkar and R. Sekar. Data Space Randomization. In *Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symp. on Information, Computer and Communications Security*, 2011.

[9] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[10] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Symp. on Operating Systems Design and Implementation*, 2006.

[11] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *ACM Symp. on Operating Systems Principles*, 2009.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conf. on Computer and Communication Security*, 2010.

[13] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointguardTM: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2003.

[14] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[15] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[16] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[17] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. *SIGPLAN Notices*, 41(6):144–157, June 2006.

[18] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Symp. on Operating Systems Design and Implementation*, 2006.

[19] E. Göktaş, E. Athanasopoulosy, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symp. on Security and Privacy*, 2014.

[20] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2012.

[21] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2013.

[22] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symp. on Security and Privacy*, 2013.

[23] Intel Architecture Instruction Set Extensions Programming Reference. `http://download-software.intel.com/sites/default/files/319433-015.pdf`, 2013.

[24] Intel. Introduction to Intel memory protection extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conf.*, 2002.

[26] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity softwar. In *Annual Computer Security Applications Conf.*, 2006.

[27] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *ACM Conf. on Programming Language Design and Implementation*, 2005.

[28] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *ACM Conf. on Programming Language Design and Implementation*, 2007.

[29] J. Li, Z. Wang, T. K. Bletsch, D. Srinivasan, M. C. Grace, and X. Jiang. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security*, 6(4):1404–1417, Dec. 2011.

[30] The LLVM compiler infrastructure. `http://llvm.org/`.

[31] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. Cryptographically enforced control flow integrity. http://arxiv.org/abs/1408.1451, Aug. 2014.

[32] S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security Symposium*, 2006.

[33] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Intl. Symp. on Computer Architecture*, 2012.

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Safety for C. In *ACM Conf. on Programming Language Design and Implementation*, 2009.

[35] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Intl. Symp. on Memory Management*, 2010.

[36] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. on Programming Languages and Systems*, 27(3):477–526, 2005.

[37] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58):`http://phrack.com/issues.html?issue=67&id=8`, Nov. 2007.

[38] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conf. on Computer and Communication Security*, 2013.

[39] B. Niu and G. Tan. Modular control-flow integrity. In *ACM Conf. on Programming Language Design and Implementation*, 2014.

[40] PaX-Team. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[41] Phoronix. Phoronix test suite. `http://www.phoronix-test-suite.com/`.

[42] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symp. on Security and Privacy*, 2010.

[43] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conf.*, 2012.

[44] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conf. on Computer and Communication Security*, 2007.

[45] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symp. on Security and Privacy*, pages 574–588, 2013.

[46] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. *IEEE Symp. on Security and Privacy*, 2013.

[47] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, 2014.

[48] A. van de Ven and I. Molnar. Exec Shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.

[49] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime intrusion prevention evaluator. In *Annual Computer Security Applications Conf.*, 2011.

[50] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symp. on Security and Privacy*, 2009.

[51] B. Zeng, G. Tan, and Ú. Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security Symposium*, 2013.

[52] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM Conf. on Computer and Communication Security*, 2011.

[53] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Symp. on Security and Privacy*, 2013.

[54] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.

Ironclad Apps: End-to-End Security via Automated Full-System Verification

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan[†], Bryan Parno, Danfeng Zhang[⋆], Brian Zill
Microsoft Research [†] University of Pennsylvania [⋆] Cornell University

Abstract

An Ironclad App lets a user securely transmit her data to a remote machine with the guarantee that every instruction executed on that machine adheres to a formal abstract specification of the app's behavior. This does more than eliminate implementation vulnerabilities such as buffer overflows, parsing errors, or data leaks; it tells the user exactly how the app will behave at all times. We provide these guarantees via complete, low-level software verification. We then use cryptography and secure hardware to enable secure channels from the verified software to remote users. To achieve such complete verification, we developed a set of new and modified tools, a collection of techniques and engineering disciplines, and a methodology focused on rapid development of verified systems software. We describe our methodology, formal results, and lessons we learned from building a full stack of verified software. That software includes a verified kernel; verified drivers; verified system and crypto libraries including SHA, HMAC, and RSA; and four Ironclad Apps.

1 Introduction

Today, when Alice submits her personal data to a remote service, she has little assurance that her data will remain secure. At best, she has vague legal guarantees provided by the service's privacy policy and the hope that the owner will follow industry best practices. Even then, a vulnerable OS, library, or application may undermine the service provider's best intentions [51].

In theory, complete formal verification of the service's code would replace this tenuous position with the strong mathematical guarantee that the service precisely matches Alice's formally specified security expectations. Unfortunately, while software verification provides strong guarantees [4, 6, 8, 17, 39], the cost is often high [25, 35, 36]; e.g., seL4 took over 22 person-years of effort to verify a microkernel. Some strong guarantees have been obtained in much less time, but those guarantees depend on unverified lower-level code. For example, past work produced a verified TLS implementation [9] and a proof of correctness for RSA-OAEP [7]. In both cases, though, they assumed the crypto libraries, their runtimes (e.g., .NET), and the OS were correct.

In contrast, we aim to create *Ironclad Apps* that are verifiably end-to-end secure, meaning that: (1) The verification covers *all* code that executes on the server, not just the app but also the OS, libraries, and drivers. Thus, it does not assume that any piece of server software is correct. (2) The proof covers the *assembly code* that gets executed, not the high-level language in which the app is written. Thus, it assumes that the hardware is correct, but assumes nothing about the correctness of the compiler or runtime. (3) The verification demonstrates *remote equivalence*: that to a remote party the app's implementation is *indistinguishable* from the app's high-level abstract state machine.

Verifiable remote equivalence dictates the behavior of the entire system in every possible situation. Thus, this approach provides stronger guarantees than type checkers or tools that look for classes of bugs such as buffer overflows or bounds errors. Our proof of remote equivalence involves proving properties of both functional correctness and information flow; we do the latter by proving *noninterference*, a relationship between two runs of the same code with different inputs.

We then show how remote equivalence can be strengthened to *secure* remote equivalence via Trusted Computing [3, 53]. Specifically, the app verifiably uses secure hardware, including a TPM [63], to convince a remote client that its public key corresponds to a private key known only to the app. The client uses the public key to establish a secure channel, thereby achieving security equivalent to direct communication with the abstractly specified app [30].

Another goal of our work is to make it feasible to build Ironclad Apps with modest developer effort. Previous efforts, such as seL4 [35] or VCC [13], took tens of person-years to verify one software layer, so verifying an entire stack using these techniques may be prohibitive. To reduce developer effort, we use state-of-the-art tools for *automated* software verification, such as Dafny [39], Boogie [4], and Z3 [17]. These tools need much less guidance from developers than interactive proof assistants used in previous work [35, 52].

However, many in the verification community worry that automated verification cannot scale to large software and that the tools' heuristics inevitably lead to unstable verification results. Indeed, we encountered these challenges, and dealt with them in multiple ways: via two new tools (§3.4); via modifications to existing verification tools to support *incremental verification*, *opaque functions*, and *automatic requirement propagation*; via software engineering disciplines like *premium functions* and *idiomatic specification*; via a *nonlinear math library* that

lets us suppress instability-inducing arithmetic heuristics; and via *provably correct libraries* for performing crypto operations and manipulating arrays of bits, bytes, and words. All these contributions support stable, automated, large-scale, end-to-end verification of systems software.

To demonstrate the feasibility of our approach, we built four Ironclad Apps, each useful as a standalone service but nevertheless compactly specifiable. For instance, our Notary app securely assigns logical timestamps to documents so they can be conclusively ordered. Our other three apps are a password hasher, a multi-user trusted counter [40], and a differentially-private database [19].

We wrote nearly all of the code from scratch, including the apps, libraries, and drivers. For the OS, we used the Verve microkernel [65], modified to support secure hardware and the Dafny language. For our four apps collectively we wrote about 6K lines of implementation and 30K lines of proof annotations. Simple benchmarks experience negligible slowdown, but unoptimized asymmetric crypto workloads slow down up to two orders of magnitude.

Since we prove that our apps conform to their specifications, we want these specs to be small. Currently, the total spec size for all our apps is 3,546 lines, satisfying our goal of a small trusted computing base (TCB).

2 Goals and Assumptions

Here we summarize Ironclad's goals, non-goals, and threat model. As a running example, we use our Notary app, which implements an abstract Notary state machine. This machine's state is an asymmetric key pair and a monotonic counter, and it signs statements assigning counter values to hashes. The crypto lets a user securely communicate with it even over an untrusted network.

2.1 Goals

Remote equivalence. Any remote party, communicating with the Ironclad App over an untrusted network, should receive the same sequence of messages as she would have received if she were communicating with the app's abstract state machine over an untrusted network. For example, the Notary app will never roll back its counter, leak its private key, sign anything other than notarizations, compute signatures incorrectly, or be susceptible to buffer overflows, integer overflows, or any other implementation-level vulnerabilities.

Secure channel. A remote user can establish a secure channel to the app. Since this protects the user's communication from the untrusted network, the remote equivalence guarantee leads to security commensurate with actual equivalence. For example, the Notary's spec says it computes its key pair using secure randomness, then obtains an attestation binding the public key and the app's code to a secure platform. This attestation convinces a remote user that a notarization signed with the corresponding private key was generated by the Notary's code, which is equivalent to the abstract Notary state machine. Note that not all messages need to use the secure channel; e.g., hashes sent to the Notary are not confidential, so the app does not expect them to be encrypted.

Completeness. Every software component must be either verified secure or run in a verified-secure sandbox; our current system always uses the former option. The assurance should cover the entire system as a coherent whole, so security cannot be undermined by incorrect assumptions about how components interact. Such gaps introduced bugs in previous verification efforts [65].

Low-level verification. Since complex tools like compilers may introduce bugs (a recent study found 325 defects in 11 C compilers [66]), we aim to verify the actual instructions that will execute rather than high-level code. Verifying assembly also has a potential performance benefit: We can hand-tune our assembly code without fear of introducing bugs that violate our guarantees.

Rapid development by systems programmers. To push verification towards commercial practicality, we need to improve the scale and functionality of verification tools to support large, real-world programs. This means that non-expert developers should be able to rapidly write and efficiently maintain verified code.

2.2 Non-goals

Compatibility. Ideally, we would verify existing code written in standard languages. However given the challenges previous efforts have faced [13], we choose to focus on fresh code written in a language designed to support verification. If we cannot achieve the goals above in such a setting, then we certainly cannot achieve it in the challenging legacy setting.

Performance. Our primary goal is to demonstrate the feasibility of verifying an entire software stack. Hence, we focus on single-core machines, poll for network packets rather than using interrupts, and choose algorithms that facilitate proofs of correctness rather than performance.

However, verification gives us a strong safety net with which to perform arbitrarily aggressive optimizations, since we can count on our tools to catch any errors that might be introduced. We exploited this repeatedly.

2.3 Threat model and assumptions

Ironclad provides security against software-based attackers, who may run arbitrary software on the machine before the Ironclad App executes and after it terminates. The adversary may compromise the platform's firmware, BIOS, and peripheral devices, such as the network card. We assume the CPU, memory, and chipset are correct, and the attacker does not mount physical attacks, such as electrically probing the memory bus.

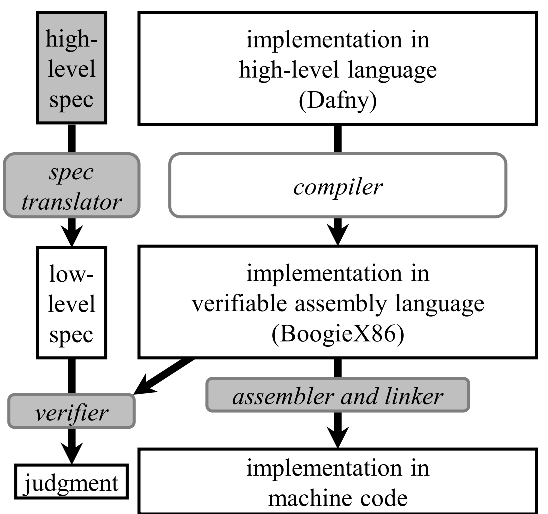


Figure 1: **Methodology Overview.** *Rounded rectangles represent tools; regular rectangles represent artifacts. Trusted components are shaded.*

We focus on privacy and integrity; we do not prove liveness, so attacks or bugs may result in denial of service. Our hardware model is currently inadequate to prove the absence of side channels due to cache or timing effects.

We assume the platform has secure hardware support, specifically a *Trusted Platform Module* (TPM). Deployed on over 500 million computers [64], the TPM provides a hardware-based root of trust [3, 53, 63]. That is, it records information about all software executed on the platform during a boot cycle in a way that can be securely reported, via an *attestation* protocol, to a remote party. The TPM maintains records about the current boot cycle in the form of hash chains maintained in volatile *Platform Configuration Registers* (PCRs). Software can add information to the PCRs via an *extend* operation. This operation updates a PCR to the hash of its previous value concatenated with the new information. The TPM also has a private RSA key that never leaves the device and can be used to *attest* to the platform's current state by signing the PCR values. The TPM's manufacturer certifies that the corresponding public key is held by a real hardware TPM, preventing impersonation by software. Finally, the TPM provides access to a stream of secure random bytes.

3 The Ironclad Methodology

This section describes our methodology for verifying Ironclad Apps are secure and for efficiently building them.

3.1 Overview

Previous verification efforts required >20 person-years of effort to develop relatively small verified software. Since we aim to perform low-level, full-system verification with modest effort, our methodology (Fig. 1) differs from previous efforts in significant ways.

With Ironclad, we use a verification stack based on Floyd-Hoare reasoning (§3.2) to prove the functional correctness of our code. We write both our specifications (§3.3) and code (§3.4) in Dafny [39], a remarkably usable high-level language designed to facilitate verification. Unlike tools used in previous efforts, Dafny supports automated verification via the Z3 [17] SMT solver, so the tool often automatically fills in low-level proof details.

Given correct Dafny code, we built automated tools to translate our code to BoogieX86 [65], a verifiable assembly language (§3.4). The entire system is verified at the assembly level using the Boogie verifier [4], so any bugs in Dafny or in the compiler will be caught at this stage.

At every stage, we use and extend existing tools and build new ones to support rapid development of verified code (§3.5), using techniques like real-time feedback in developer UIs and multi-level verification result caching.

Finally, since many security properties cannot be expressed via functional correctness, we develop techniques for verifying *relational properties* of our code (§3.6).

If all verification checks pass, a simple trusted assembler and linker produces the machine code that actually runs. We run that code using the platform's secure late-launch feature (§6.1), which puts the platform into a known-good state, records a hash of the code in the TPM (§2.3), then starts executing verified code. These steps allow remote parties to verify that Ironclad code was indeed properly loaded, and they prevent any code that runs before Ironclad, including the boot loader, from interfering with its execution.

3.2 Background: Floyd-Hoare verification

We verify Ironclad Apps using Floyd-Hoare reasoning [21, 31]. In this approach, programs are annotated with assertions about program state, and the verification process proves that the assertions will be valid when the program is run, for all possible inputs. As a simple example, the following program is annotated with an assertion about the program state at the end of a method (a "postcondition"), saying that the method output O must be an even number:

```
method Main(S, I) returns(O)
    ensures even(O);
{ O := (S + S) + (I + I); }
```

A tool like Dafny or Boogie can easily and automatically verify that the postcondition above holds for all possible inputs S and I.

For a long-running program with multiple outputs, we can specify a restriction on *all* of the program's outputs by annotating its output method with a precondition. For instance, writing:

```
method WriteOutput(O)   // Trusted output
    requires even(O);   // method
```

ensures that the verifier will reject code unless, like the following, it can be proven to only output even numbers:

```
method Main() {
  var count := 0;
  while(true) invariant even(count) {
    count := count + 2;
    WriteOutput(count);
  } }
```

Boogie and Dafny are sound, i.e., they will never approve an incorrect program, so they cannot be complete, i.e., they will sometimes fail to automatically recognize valid programs as correct. Thus, they typically require many preconditions, postconditions, and loop invariants inside the program to help them complete the verification, in addition to the preconditions and postconditions used to write the trusted specifications. The loop invariant `invariant even(count)` in the example above illustrates this: it is not part of the trusted specification, but instead serves as a hint to the verification tool.

By itself, Floyd-Hoare reasoning proves safety properties but not liveness properties. For example, a postcondition establishes a property of the state upon method exit, but the method may fail to terminate. We have not proven liveness for Ironclad Apps.

3.3 Writing trustworthy specifications

To build Ironclad Apps, we write two main types of specifications: hardware and apps. For hardware specs, since we aim for low-level verification, we write a specification for each of the ∼56 assembly instructions our implementation will use. An instruction's spec describes its preconditions and its effects on the system. For example, `Add` ensures that the sum of the input registers is written to the destination register, and requires that the input values not cause the sum to overflow.

For app specs, we write abstract descriptions of desired app behavior. These are written modularly in terms of lower-level library specs. For example, the spec for the Notary describes how the app's state machine advances and the outputs permitted in each state; one possible output is a signed message which is defined in terms of our spec for RSA signing.

The verification process removes all implementation code from the TCB by proving that it meets its high-level spec given the low-level machine spec. However, the specs themselves *are* part of the TCB, so it is crucial that they be worthy of users' trust. To this end, we use *spec-first design*, *idiomatic specification*, and *spec reviews*.

Spec-first design. To encourage spec quality, we write each specification before starting on its implementation. This order makes the spec likely to express desired properties rather than a particular mechanism. Writing the spec afterwards might port implementation bugs to the spec.

Idiomatic specification. To ensure trustworthy specs, we aim to keep them small and simple, making bugs less likely and easier to spot. We accomplish this by specifying only the feature subset that our system needs, and by ensuring that the implementation cannot trigger other features; e.g., our verifier will not permit any assembly instructions not in the hardware spec. This is crucial for devices; e.g., the TPM's documentation runs to hundreds of pages, but we need only a fraction of its functionality. Hence, our TPM spec is only 296 source lines of code (SLOC).

Spec reviews. We had two or more team members develop each spec, and another review their work independently. This caught several bugs before writing any code.

Despite our techniques, specs may still contain bugs. However, we expect them to contain significantly fewer bugs than implementations. First, our specs are smaller (§8.1). Second, our specs are written in a more abstract, declarative fashion than implementation code, making spec bugs both less likely to occur and easier to find when they do occur. For example, one line in our Notary spec (§5.1) says that a number representing a counter is incremented. The code *implementing* that addition, in contrast, involves hundreds of lines of code: it implements the unbounded-precision number using an array of machine words, so addition must handle carries and overflow.

Overall, our experience (§7) suggests specs are indeed more trustworthy than code.

3.4 Producing verifiable assembly language

To enable rapid, large-scale software development while still verifying code at a low level, we take a two-layer verification approach (Figure 1): we write our specs and implementation in the high-level Dafny language, but we re-verify the code after compiling to assembly language.

We replaced the existing Dafny compiler targeting .NET and Windows with two new components, a trusted spec translator and a new untrusted compiler called DafnyCC. The trusted spec translator converts a tiny subset of Dafny into BoogieX86. This subset includes just those features useful in writing specs: e.g., functions, type definitions, and sequences, but not arrays.

Our untrusted DafnyCC compiler, in contrast, consumes a large subset of the Dafny language. It translates both the code *and* the proofs written in Dafny into BoogieX86 assembly that Boogie can automatically verify. It also automatically inserts low-level proofs that the stack is used safely (§6.3), that OS invariants are maintained (§6.4), etc. Because all of the code emitted by DafnyCC is verified by Boogie, none of its complexity is trusted. Thus, we can add arbitrarily complex features and optimizations without hurting security. Indeed, Boogie caught several bugs made during compilation (§7.7).

3.5 Rapid verification

A key goal of Ironclad is to reduce the verification burden for developers, so we use the following techniques to support rapid verification.

Preliminary verification. Although ultimately we must verify code at the assembly level, it is useful to perform a fast, preliminary verification at the Dafny level. This lets the developer quickly discover bugs and missing proof annotations. The verification is particularly rapid because Dafny includes a plugin for the Visual Studio interactive development environment that verifies code incrementally as the developer types, emitting error messages and marking the offending code with squiggly underlines.

Modular verification. We added support to Dafny for modular verification, allowing one file to import another file's interfaces without reverifying that code.

Shared verification. Our *IronBuild* tool shares verification results among developers via a cloud store. Since each developer verifies code before checking it in, whenever another developer checks out code, verification will succeed immediately based on cached results. IronBuild precisely tracks dependencies by hash to ensure fidelity.

3.6 Verifying relational properties

For Ironclad Apps, we prove properties beyond functional correctness, e.g., that the apps do not leak secrets such as keys. Although standard Floyd-Hoare tools like Boogie and Dafny focus on functional correctness, we observed that we could repurpose a Boogie-based experimental tool, SymDiff [37], to prove *noninterference* properties. We combine these proofs with our functional correctness proofs to reason about the system's security (§4).

Suppose that variable `S` represents a secret inside the program and `I` represents a public input to the program. The statement $\texttt{O} := (\texttt{S}+\texttt{S})+(\texttt{I}+\texttt{I})$ satisfies a functional correctness specification `even(O)`. However, in doing so, it leaks information about the secret `S`.

The statement $\texttt{O} := (\texttt{S}-\texttt{S})+(\texttt{I}+\texttt{I})$, by contrast, satisfies `even(O)` yet leaks no information about `S`. Intuitively, the value stored in `O` depends on `I` but is independent of `S`. The concept of noninterference [24, 57, 61] formalizes this intuition by reasoning about multiple executions of a program, and comparing the outputs to see which values they depend on. Suppose that we pass the same public input `I` to all the executions, but vary the secret `S` between the executions. If all the executions produce the same output `O` regardless of `S`, then `O` is independent of `S`, and the program leaks nothing about `S`.

Mathematically, noninterference means that for all possible pairs of executions, if the public inputs `I` are equal but the secrets `S` may be different, then the outputs `O` are equal. (Some definitions also require that termination is independent of secrets [57], while others do not [61]; for simplicity, we use the latter.) More formally, if we call the two executions in each pair L and R, for left and right, then noninterference means $\forall \texttt{S}_L, \texttt{S}_R\ .\ \texttt{I}_L = \texttt{I}_R \Longrightarrow \texttt{O}_L = \texttt{O}_R$. For instance, $\texttt{O} := (\texttt{S}-\texttt{S})+(\texttt{I}+\texttt{I})$ satisfies this condition, but $\texttt{O} := (\texttt{S}+\texttt{S})+(\texttt{I}+\texttt{I})$ does not.

To allow the SymDiff tool to check noninterference, we annotate some of our code with explicit relational annotations [5], writing x_L as `left`(x) and x_R as `right`(x):

```
method Test(S, I) returns(O)
    requires left(I) == right(I);
    ensures left(O) == right(O);
    ensures even(O);
{ O := (S - S) + (I + I); }
```

The relational precondition `left(I) == right(I)` means SymDiff must check that $\texttt{I}_L = \texttt{I}_R$ wherever `Test` is called, and the relational postcondition `left(O) == right(O)` means SymDiff must check that this method ensures $\texttt{I}_L = \texttt{I}_R \Longrightarrow \texttt{O}_L = \texttt{O}_R$.

However, for most of our code, SymDiff leverages our existing functional correctness annotations and does not need relational annotations. For example, SymDiff needs only the functional postcondition in this code:

```
method ComputeIpChecksum(I) returns(O)
    ensures O == IpChecksum(I);
```

to infer that if $\texttt{I}_L = \texttt{I}_R$, then $\texttt{IpChecksum}(\texttt{I}_L) = \texttt{IpChecksum}(\texttt{I}_R)$, so $\texttt{O}_L = \texttt{O}_R$.

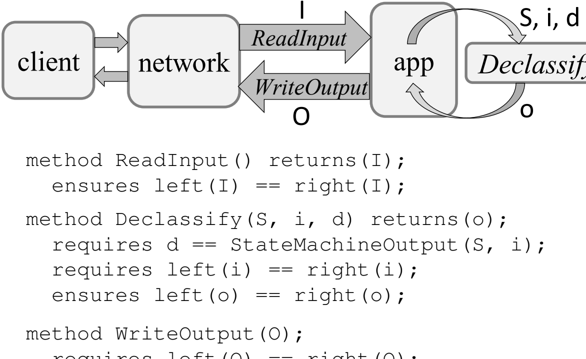
4 Proving Ironclad Security Properties

This section describes how we combine the previous section's ideas of functional correctness, like `even(O)`, and noninterference, like $\texttt{I}_L = \texttt{I}_R \Longrightarrow \texttt{O}_L = \texttt{O}_R$, to prove the security of our Ironclad Apps. It describes the architecture, theorems, and proofs at a high level. In §5, we show how they are instantiated for each app, and in §6 we give details about the key lemmas we prove about our system to support these high-level results.

4.1 Declassification and the Ironclad architecture

Pure noninterference establishes that a program's output values are completely independent of the program's secrets, but this requirement is too strong for most real-world systems. In practice, programs deliberately allow limited influence of the secrets on the output, such as using a secret key to sign an output. A security policy for such programs explicitly *declassifies* certain values, like a signature, so they can be output despite being dependent on secrets.

Figure 2 shows the overall structure of the Ironclad system, including an abstract declassifier that authorizes the release of selected outputs derived from secrets. We express each app's declassification policy as a state machine, thereby binding the release of secret-derived data to the high-level behavior of the abstract app specification. We assume that the client communicates with the Ironclad App across a network that may drop, delay, duplicate, or mangle data. The network, however, does not have access to the app's secrets. The app receives some possibly-mangled inputs `I` and responds by sending some outputs `O` to the network, which may mangle `O` before passing them to the client. While computing the outputs `O`, the app may




```
method ReadInput() returns(I);
  ensures left(I) == right(I);
method Declassify(S, i, d) returns(o);
  requires d == StateMachineOutput(S, i);
  requires left(i) == right(i);
  ensures left(o) == right(o);
method WriteOutput(O);
  requires left(O) == right(O);
```

Figure 2: **Abstract system structure and trusted input/output/declassify specification.**

appeal to the declassification policy as many times as it wishes. Each time, it passes its secrets $\mathtt{S}$, some inputs $\mathtt{i}$, and the desired declassified outputs $\mathtt{d}$ to the declassifier. For verification to succeed, the desired outputs must equal the outputs according to the abstract state machine's policy: $\mathtt{d} = \mathit{StateMachineOutput}(\mathtt{S}, \mathtt{i})$. If static verification proves that the declassification policy is satisfied, the declassifier produces declassified outputs $\mathtt{o}$ that the app can use as part of its outputs $\mathtt{O}$.

In the real implementation, $\mathtt{o}$ simply equals $\mathtt{d}$, so that the declassifier is a no-op at run-time. Nevertheless, we hide this from the verifier, because we want to reveal $\mathtt{o}_L = \mathtt{o}_R$ without revealing $\mathtt{d}_L = \mathtt{d}_R$; in some cases where the secrets $\mathtt{S}$ are in principle computable by brute-force search on $\mathtt{d}$ (e.g., by factoring an RSA public key), $\mathtt{d}_L = \mathtt{d}_R$ might imply $\mathtt{S}_L = \mathtt{S}_R$, which we do not want.

4.2 Ironclad security theorems

Given the execution model described above, for each of our apps, we first prove functional correctness as a precondition for declassification:

Theorem 1 FUNCTIONAL CORRECTNESS. *At each declassification Declassify(S, i, d), the desired outputs d satisfy the app's functional correctness policy, according to the app's abstract state machine: $d = \mathit{StateMachineOutput}(S, i)$.*

In other words, we only declassify values that the abstract state machine would have output; the state machine clearly considers these values safe to output.

Second, we split noninterference into two parts and prove both: noninterference along the path from the inputs $\mathtt{I}$ to the declassifier, and noninterference along the path from the declassifier to the outputs $\mathtt{O}$:

Theorem 2 INPUT NONINTERFERENCE. *At each declassification Declassify(S, i, d), $I_L = I_R \Longrightarrow i_L = i_R$.*

In other words, the declassifier's public inputs $\mathtt{i}$ may depend on inputs from the network $\mathtt{I}$, but not on secrets $\mathtt{S}$.

Theorem 3 OUTPUT NONINTERFERENCE. *Each time the program outputs O, $I_L = I_R \wedge o_L = o_R \Longrightarrow O_L = O_R$.*

In other words, the outputs $\mathtt{O}$ may depend on inputs from the network $\mathtt{I}$ and on any declassified values $\mathtt{o}$, but not on secrets $\mathtt{S}$.

As discussed in more detail in later sections, we carried out formal, mechanized proofs of these three theorems using the Boogie and SymDiff tools for each Ironclad App.

These theorems imply remote equivalence:

Corollary 1 REMOTE EQUIVALENCE. *To a remote party, the outputs received directly from the Ironclad App are equal to the outputs generated by the specified abstract state machine over some untrusted network, where the state machine has access to the trusted platform's secrets, but the untrusted network does not. (Specifically, if the Ironclad App generates some outputs O_L and the untrusted network generates some outputs O_R, then $O_L = O_R$.)*

Proof Sketch: We prove this by constructing an alternative, abstract counterpart to the Ironclad App. Label the real Ironclad App L and the counterpart R. The counterpart R consists of two components: the specified abstract state machine, which can read the trusted platform's secrets $\mathtt{S}$, and an untrusted network, which cannot. We construct R by using the actual Ironclad App code as the untrusted network, with two changes. First, in the untrusted network, we replace the real secrets $\mathtt{S}$ with an arbitrary value $\mathtt{S'}$, modeling the network's lack of access to the real secrets $\mathtt{S}$. Second, we replace R's ordinary declassifier with the abstract state machine producing $\mathtt{o}_R = \mathit{StateMachineOutput}(\mathtt{S}, \mathtt{i}_R)$, modeling the abstract state machine's access to the real secrets $\mathtt{S}$. We pass the same input to both the real Ironclad App L and to R, so that $\mathtt{I}_L = \mathtt{I}_R$. By INPUT NONINTERFERENCE, the inputs to the declassifier are the same: $\mathtt{i}_L = \mathtt{i}_R$. The real declassifier simply returns $\mathtt{o}_L = \mathtt{d}_L$, and by FUNCTIONAL CORRECTNESS, the real Ironclad App produces the outputs $\mathtt{d}_L = \mathit{StateMachineOutput}(\mathtt{S}, \mathtt{i}_L)$. Since $\mathtt{i}_L = \mathtt{i}_R$ and we pass the same secrets $\mathtt{S}$ to both L and R, we conclude that $\mathtt{o}_L = \mathtt{d}_L = \mathit{StateMachineOutput}(\mathtt{S}, \mathtt{i}_L) = \mathit{StateMachineOutput}(\mathtt{S}, \mathtt{i}_R) = \mathtt{o}_R$. Then by OUTPUT NONINTERFERENCE, both L and R generate the same outputs: $\mathtt{O}_L = \mathtt{O}_R$. ■

This shows that the Ironclad App's output is the same as that of the abstract state machine and an untrusted network. Thus, the output a remote party sees, which is produced by the Ironclad App and an *actual* untrusted network, is the same as that of the abstract state machine and an untrusted network composed of the actual and chosen untrusted networks.

4.3 Limitations of this model

Since we have not formally proven liveness properties like termination, an observer could in principle learn informa-

```
datatype NotaryState = NotaryState_c(
  keys:RSAKeyPair, cntr:nat);

predicate NotarizeOpCorrect(
  in_st:NotaryState, out_st:NotaryState,
  in_msg:seq<int>, out_stmt:seq<int>,
  out_sig:seq<int>)
{
 ByteSeq(in_msg)
 && out_st.keys == in_st.keys
 && out_st.cntr == in_st.cntr + 1
 && out_stmt==[OP_COUNTER_ADV]
    + rfc4251_encode(out_st.cntr) + in_msg
 && out_sig==RSASign(in_st.keys, out_stmt)
}
```

Figure 3: **Part of the Notary Spec.** *Simplified for brevity and clarity, this is a predicate the implementation must satisfy before being allowed to declassify `out_sig`, which otherwise cannot be output because it depends on secret data.*

tion about the secrets from whether an output was generated for a given input. Also, we have not formally proved timing properties, so an observer could also learn information from a timing channel. To eliminate the possibility of such timing-based information leakages, in the future we would like to prove that the time of the outputs is independent of secrets. The literature contains many possible approaches [10, 15, 26]; for example, we might prove an upper bound on the time taken to produce an output, and delay each output until the upper bound is reached.

5 Ironclad Applications

To make the guarantees of remote equivalence concrete, we describe the four apps we built. The proof for each app, in turn, builds on lemmas about lower-level libraries, drivers, and OS, which we discuss in §6.

Each app compiles to a standalone system image that communicates with other machines via UDP. Nevertheless, each is a useful complete application that would merit at least one dedicated machine in a data center. In the future, hardware support for fine-grained secure execution environments [42] may offer a simple path towards multiplexing Ironclad Apps.

5.1 Notary

Our Notary app securely assigns logical timestamps to documents so they can be conclusively ordered. This is useful, e.g., for establishing patent priority [28] or conducting online auctions [62]. Typically, users of such a service must trust that some machine is executing correct software, or that at least k of n machines are [12]. Our Ironclad Notary app requires no such assumption.

Lemma 1 NOTARY REMOTE EQUIVALENCE. *The Notary app is remotely equivalent to a state machine with the following state:*

- $\langle \mathsf{PublicKey}, \mathsf{PrivateKey} \rangle$, *computed using the RSA key generation algorithm from the first consecutive sequence of random bytes read from the TPM;*
- *a TPM, whose PCR 19 has been extended with the public part of that key pair; and*
- *a* Counter, *initialized to 0;*

and the following transitions:

- *Given input* $\langle \texttt{connect}, \mathsf{Nonce} \rangle$, *it changes the TPM state by obtaining a quote* Quote *over PCRs 17–19 and external nonce* Nonce. *It then outputs* $\langle \mathsf{PublicKey}, \mathsf{Quote} \rangle$.
- *Given input* $\langle \texttt{notarize}, \mathsf{Hash} \rangle$, *it increments* Counter *and returns* $\mathsf{Sig}_{\mathsf{PrivateKey}}(\mathit{OP\text{-}CTR\text{-}ADV} \,\|\, \mathsf{RFC4251Encode}(\mathsf{Counter}) \,\|\, \mathsf{Hash})$.

Figure 3 shows part of the corresponding Dafny spec.

Proving this lemma required proofs of the following. (1) Input non-interference: the nonce and message the app passes the declassifier are based solely on public data. (2) Functional correctness of `connect`: the app derives the key from randomness correctly, and the TPM quote the app obtains comes from the TPM when its PCRs are in the required state. (3) Functional correctness of `notarize`: the app increments the counter and computes the signature correctly. (4) Output non-interference: Writes to unprotected memory depend only on public data and the computed state machine outputs.

Proving remote-equivalence lemmas for the other apps, which we describe next, required a similar approach.

5.2 TrInc

Our trusted incrementer app, based on TrInc [40], generalizes Notary. It maintains per-user counters, so each user can ensure there are no gaps between consecutive values. It is a versatile tool in distributed systems, useful e.g. for tamper-resistant audit logs, Byzantine-fault-tolerant replicated state machines, and verifying that an untrusted file server behaves correctly.

Lemma 2 TRINC REMOTE EQUIVALENCE. *The TrInc app is remotely equivalent to a state machine like Notary's except that it has multiple counters, each a tuple* $\langle K_i, v_i \rangle$, *and a meta-counter initially set to 0. In place of the* `notarize` *transition it has:*

- *Given input* $\langle \texttt{create}, K \rangle$, *it sets* $i := \mathit{meta_counter}$, *increments* $\mathit{meta_counter}$, *and sets* $\langle K_i, v_i \rangle = \langle K, 0 \rangle$.
- *Given input* $\langle \texttt{advance}, i, v_{new}, \mathsf{Msg}, \mathsf{UserSig} \rangle$, *let* $v_{old} = v_i$ *in counter tuple* i. *If* $v_{old} \leq v_{new}$ *and* $\mathsf{VerifySig}_{K_i}(v_{new} \,\|\, \mathsf{Msg}, \mathsf{UserSig})$ *succeeds, it sets* $v_i := v_{new}$ *and outputs* $\mathsf{Sig}_{\mathsf{PrivateKey}}(\mathit{OP\text{-}CTR\text{-}ADV} \,\|\, \mathsf{encode}(i) \,\|\, \mathsf{encode}(v_{old}) \,\|\, \mathsf{encode}(v_{new}) \,\|\, \mathsf{Msg})$.

5.3 Password hasher

Our next app is a password-hashing appliance that renders harmless the loss of a password database. Today, attackers frequently steal such databases and mount offline attacks.

Even when a database is properly hashed and salted, low-entropy passwords make it vulnerable: one study recovered 47–79% of passwords from low-value services, and 44% of passwords from a high-value service [41].

Lemma 3 PASSHASH REMOTE EQUIVALENCE. *The PassHash app is remotely equivalent to the following state machine. Its state consists of a byte string* Secret*, initialized to the first 32 random bytes read from the TPM. Given input* $\langle \mathtt{hash}, \mathsf{Salt}, \mathsf{Password} \rangle$*, it outputs* $\mathsf{SHA256}(\mathsf{Secret} \,\|\, \mathsf{Salt} \,\|\, \mathsf{Password})$.

Meeting this spec ensures the hashes are useless to an offline attacker: Without the secret, a brute-force guessing attack on even the low-entropy passwords is infeasible.

5.4 Differential-privacy service

As an example of a larger app with a more abstract spec, we built an app that collects sensitive data from contributors and allows analysts to study the aggregate database. It guarantees each contributor *differential privacy* [19]: the answers provided to the analyst are virtually indistinguishable from those that would have been provided if the contributor's data were omitted. Machine-checked proofs are especially valuable here; prior work [46] showed that implementations are prone to devastating flaws.

Our app satisfies Dwork's formal definition: An algorithm $\mathcal{A}$ is differentially private with privacy ε if, for any set of answers $\mathcal{S}$ and any pair of databases D_1 and D_2 that differ by a single row, $P[\mathcal{A}(D_1) \in \mathcal{S}] \leq \lambda \cdot P[\mathcal{A}(D_2) \in \mathcal{S}]$, where we use the privacy parameter $\lambda = e^{\varepsilon}$ [23].

Privacy budget. Multiple queries with small privacy parameters are equivalent to a single query with the product of the parameters. Hence we use a *privacy budget* [20]. Beginning with the budget $b = \lambda$ guaranteed to contributors, each query Q with parameter λ_Q divides the budget $b' := b/\lambda_Q$; a query with $\lambda_Q > b$ is rejected.

Noise computation. We follow the model of Dwork et al. [20]. We first calculate Δ, the *sensitivity* of the query, as the most the query result can change if a single database row changes. The analyst receives the sum of the true answer and a random noise value drawn from a distribution parameterized by Δ.

Dwork et al.'s original algorithm uses noise from a Laplace distribution [20]. Computing this distribution involves computing a natural logarithm, so it cannot be done precisely on real hardware. Thus, practical implementations simulate this real-valued distribution with approximate floating point values. Unfortunately, Mironov [46] devised a devastating attack that exploits information revealed by error in low-order bits to reveal the *entire database*, and showed that all five of the main differential-privacy implementations were vulnerable.

To avoid this gap between proof and implementation, we instead use a noise distribution that only involves rational numbers, and thus can be sampled precisely using

```
predicate DBsSimilar(d1:seq<Row>,d2:seq<Row>)
  |d1| == |d2| &&
  exists diff_row ::
    forall i :: 0 <= i < |d1| && i != diff_row
     ==> d1[i] == d2[i]

predicate SensitivitySatisfied(prog:seq<Op>,
               min:int, max:int, delta:int)
  forall d1:seq<Row>, d2:seq<Row> ::
  Valid(d1)&&Valid(d2)&&DBsSimilar(d1, d2)==>
  -delta <= MapperSum(d1, prog, min, max) -
            MapperSum(d2, prog, min, max)
         <= delta
```

Figure 4: **Summing Reducer Sensitivity.** *Our differential-privacy app is verified to satisfy a predicate like this, relating reducer output sensitivity to the* Δ *used in noise generation.*

the x86 instruction set. In our specification, we model these rational numbers with real-valued variables, making the spec clearer and more compact. We then prove that our 32-bit-integer-based implementation meets this spec.

Lemma 4 DIFFPRIV REMOTE EQUIVALENCE. *The DiffPriv app is remotely equivalent to a state machine with the following state:*

- *key pair and TPM initialized as in Notary;*
- *remaining budget b, a real number; and*
- *a sequence of rows, each consisting of a duplicate-detection nonce and a list of integer column values;*

and with transitions that connect to the app, initialize the database, add a row, and perform a query.

We also prove a higher-level property about this app:

Lemma 5 SENSITIVITY. *The value* Δ *used as the sensitivity parameter in the spec's noise computation formula is the actual sensitivity of the query result. That is, if we define* $\mathcal{A}(D)$ *as the answer the app computes when the database is D, then for any two databases* D_1 *and* D_2*,* $|\mathcal{A}(D_1) - \mathcal{A}(D_2)| \leq \Delta$.

To make this verifiable, we use Airavat-style queries [56]. That is, each query is a *mapper*, which transforms a row into a single value, and a *reducer*, which aggregates the resulting set; only the latter affects sensitivity. The analyst can provide an arbitrary mapper; we provide, and prove sensitivity properties for, the single reducer `sum`. It takes RowMin and RowMax parameters, clipping each mapper output value to this range. Figure 4 shows the property we verified: that the sensitivity of `sum` is $\Delta = \mathsf{RowMax} - \mathsf{RowMin}$ regardless of its mapper-provided inputs.

6 Full-System Verification

We have mechanically verified the high-level theorems described in §4. Although the mechanical verification uses automated theorem proving, the code must contain manual annotations, such as loop invariants, preconditions, and postconditions (§3.2). One can think of these

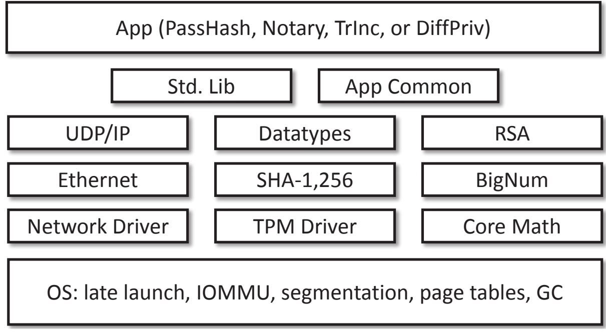


Figure 5: **System Overview.**

annotations, spread throughout the code, as lemmas that build to the final high-level theorems.

To convey the work necessary to complete the verification, this section gives a sampling of the key lemmas we proved along the way. For clarity and conciseness, we state each lemma as brief English text; the real mechanical "lemmas" are the annotations in the code itself. The lemmas described in this section are not, on their own, sufficient for the proof, since they are only a sampling. Nevertheless, a failure in any of the lemmas below would cause the high-level theorems to fail; we would not be able to establish the overall correctness of an Ironclad App if, for example, the cryptographic library or the garbage collector were incorrect.

6.1 Memory, devices, and information flow

Lemma 6 IOMMU CONFIGURATION. *The Ironclad Apps configure the IOMMU to divide memory into device-accessible and app-private memory; non-device operations access only app-private memory.*

Our assembly language instruction specifications check that non-device memory operations only access app-private memory that has been protected by the hardware's device exclusion vector, a simple IOMMU.

Commodity CPUs from AMD [1] and Intel [32] provide a *dynamic root-of-trust for measurement* (DRTM) feature, a.k.a. *late launch* [53]. It resets the CPU to a known state, stores a measurement (hash) of the in-memory code pointed to by the instruction's argument, and jumps to that code. After a late launch, the hardware provides the program control of the CPU and 64 KiB of protected memory. To use more than 64 KiB, it must first extend the IOMMU's protections, using our specification for IOMMU configuration. Only then can the program satisfy the preconditions for assembly language instructions accessing memory outside the 64-KiB region.

Lemma 7 DEVICES SEE NO SECRETS. *Only non-secret data is passed to devices.*

Our assembly language instruction specifications require that stores to device-accessible memory, i.e., memory that the IOMMU allows devices to see, can only store non-secret data $\circ$. In §3.6's terminology, non-secret means that $\circ_L = \circ_R$. More specifically, we require that the left and right executions generate the same sequence of device stores: the same values to the same addresses, modulo timing and liveness.

To prove $\circ_L = \circ_R$, we annotate our implementation's input and output paths with relational annotations. These input and output paths include the application event loops and the networking stack. For example, the Ethernet, IP, and UDP layers maintain relational properties on packets.

Lemma 8 KEY IN TPM. *Apps correctly extend a public key into the TPM's PCR 19. The private key is generated using TPM randomness and never leaves the platform.*

Lemma 9 ATTESTATION. *Apps generate a correct TPM attestation after extending their public key into a PCR.*

Corollary 2 SECURE CHANNEL. *If a remote client receives a public key and an attestation, and the attested PCR code values (PCRs 17, 18) match those of an Ironclad App, and the attested PCR data values (PCR 19) match the public key, and a certificate shows the attestation is from a legitimate hardware TPM manufacturer, then the client can use the public key to establish a secure channel directly to the Ironclad App.*

6.2 Cryptographic libraries

Lemma 10 HASHING. *Our SHA-{1,256} conforms to FIPS 180-4 [50], and our HMAC to FIPS 198-1 [49].*

Lemma 11 RSA OPERATIONS. *RSA keys are generated using consecutive randomness from the TPM (not selectively sampled), and pass the Miller-Rabin primeness test [45, 54]. Our implementations of RSA encrypt, decrypt, sign, and verify, including padding, produce byte arrays that conform to PKCS 1.5 and RSA standards [33].*

For basic cryptographic primitives such as hash functions, functional correctness is the best we can hope to verify. For instance, there is no known way to prove that SHA-256 is collision-resistant.

The RSA spec, derived from RFC 2313 [33], defines encryption and signature operations as modular exponentiation on keys made of Dafny's ideal integers. The key-generation spec requires that the key be made from two random primes.

To implement these crypto primitives, we built a BigNum library. It implements arbitrary-precision integers using arrays of 32-bit words, providing operations like division and modulo needed for RSA. BigRat extends it to rationals, needed for differential privacy.

Lemma 12 BIGNUM/BIGRAT CORRECTNESS. *Each BigNum/BigRat operation produces a value representing the correct infinite-precision integer or real number.*

6.3 DafnyCC-generated code

Since the DafnyCC compiler sits outside our TCB, we have to verify the assembly language code it generates. This verification rests on several invariants maintained by all DafnyCC-generated code:

Lemma 13 TYPE SAFETY. *The contents of every value and heap object faithfully represent the expected contents according to Dafny's type system, so that operations on these values never cause run-time type errors.*

Lemma 14 ARRAY BOUNDS SAFETY. *All array operations use an index within the bounds of the array.*

Lemma 15 TRANSITIVE STACK SAFETY. *When calling a method, enough stack space remains for all stack operations in that method and those it in turn calls.*

Dafny is a type-safe language, but we cannot simply assume that DafnyCC preserves Dafny's type safety. Thus, we must prove type safety at the assembly language level by establishing typing invariants on all data structures that represent Dafny values. For example, all pointers in data structures point only to values of the expected type, and arbitrary integers cannot be used as pointers. These typing invariants are maintained throughout the Ironclad assembly language code (they appear in nearly all loop invariants, preconditions, and postconditions). In contrast to the original Verve OS [65], Ironclad does not rely on an external typed assembly language checker to check compiled code; this gives Ironclad the advantage of using a single verification process for both hand-written assembly language code and compiled code, ensuring that there are no mismatches in the verification process.

Lemma 16 HIGH-LEVEL PROPERTY PRESERVATION. *Every method proves that output stack state and registers satisfy the high-level Dafny postconditions given the high-level Dafny preconditions.*

DafnyCC maintains all Dafny-level annotations, including preconditions, postconditions, and loop invariants. Furthermore, it connects these high-level annotations to low-level stack and register values, so that the operations on stack and register values ultimately satisfy the Dafny program's high-level correctness theorems.

6.4 Maintaining OS internal invariants

Although Ironclad builds on the original Verve OS [65], we made many modifications to the Verve code to accommodate DafnyCC, the late launch process and the IOMMU (§6.1), the TPM (§2.3), segmentation, and other aspects of Ironclad. Thus, we had to prove that these modifications did not introduce any bugs into the Verve code.

Lemma 17 OPERATING SYSTEM INVARIANTS. *All operating system data structure invariants are maintained.*

Lemma 18 GARBAGE COLLECTION CORRECTNESS. *The memory manager's representation of Dafny objects correctly represents the high-level Dafny semantics.*

We modified the original Verve copying garbage collector's object representation to accommodate DafnyCC-generated code. This involved reproving the GC correctness lemma: that the GC always maintains correct object data, and never leaves dangling pointers, even as it moves objects around in memory. Our modification initially contained a design flaw in the object header word: we accidentally used the same bit pattern to represent two different object states, which would have caused severe and difficult-to-debug memory corruption. Verification found the error in seconds, before we ran the new GC code.

7 Experiences and Lessons Learned

In this section, we describe our experiences using modern verification tools in a large-scale systems project, and the solutions we devised to the problems we encountered.

7.1 Verification automation varies by theory

Automated theorem provers like Z3 support a variety of theories: arithmetic, functions, arrays, etc. We found that Z3 was generally fast, reliable, and completely automated at reasoning about addition, subtraction, multiplication/division/mod by small constants, comparison, function declarations, non-recursive function definitions, sequence/array subscripting, and sequence/array updates. Z3 sometimes needed hints to verify sequence concatenation, forall/exists, and recursive function definitions, and to maintain array state across method invocations.

Unfortunately, we found Z3's theory of nonlinear arithmetic to be slow and unstable; small code changes often caused unpredictable verification failures (§7.2).

7.2 Verification needs some manual control

As discussed in §1, verification projects often avoid automated tools for fear that such tools will be unstable and/or too slow to scale to large, complex systems. Indeed, we encountered verification instability for large formulas and nonlinear arithmetic. Nevertheless, we were able to address these issues by using modular verification (§3.5), which reduced the size of components to be verified, and two additional solutions:

Opaque functions. Z3 may unwrap function definitions too aggressively, each time obtaining a new fact, often leading to timeouts for large code. To alleviate this, we modified Dafny so a programmer can designate a function as *opaque*. This tells the verifier to ignore the body, except in places where the programmer explicitly indicates.

Nonlinear math library. Statements about nonlinear integer arithmetic, such as $\forall x,y,z : x(y+z) = xy+xz$, are not, in general, decidable [17]. So, Z3 includes heuristics for reasoning about them. Unfortunately, if a complicated

method includes a nonlinear expression, Z3 has many options for applicable heuristics, leading to instability.

Thus, we disable Z3's nonlinear heuristics, except on a few files where we prove simple, fundamental lemmas, such as $(x > 0 \land y > 0) \Rightarrow xy > 0$. We used those fundamental lemmas to prove a library of math lemmas, including commutativity, associativity, distributivity, GCDs, rounding, exponentiation, and powers of two.

7.3 Existing tools make simple specs difficult

To enhance the security of Ironclad Apps, we aim to minimize our TCB, particularly the specifications.

Unfortunately, Dafny's verifier insists on proving that, whenever one function invokes another, the caller meets the callee's pre-conditions. So, the natural spec for SHA,

```
function SHA(bits:seq<int>):seq<int>
  requires |bits|<power2(64);
{ .... }
function SHA_B(bytes:seq<int>):seq<int>
{ SHA(Bytes2Bits(bytes)) }
```

has a problem: the call from SHA_B to SHA may pass a bit sequence whose length is $\geq 2^{64}$.

We could fix this by adding

```
requires |bytes|<power2(61);
```

to SHA_B, but this is insufficient because the verifier needs help to deduce that 2^{61} bytes is 2^{64} bits. So we would also have to embed a mathematical proof of this in the body of SHA_B, leading to a bloated spec.

Automatic requirements. Our solution is to add *automatic requirement propagation* to Dafny: A spec writer can designate a function as autoReq, telling Dafny to automatically add pre-conditions allowing it to satisfy the requirements of its callees. For instance, if we do this to SHA_B, Dafny gives it the additional pre-condition:

```
requires |Bytes2Bits(bytes)|<power2(64);
```

This makes the spec verifiable despite its brevity.

Premium functions. Our emphasis on spec simplicity can make the implementor's job difficult. First, using autoReq means that the implementor must satisfy a pile of ugly, implicit, machine-generated pre-conditions everywhere a spec function is mentioned. Second, the spec typically contains few useful post-conditions because they would bloat the spec. For instance, SHA does not state that its output is a sequence of eight 32-bit words.

We thus introduce a new discipline of using *premium* functions in the implementation. A premium function is a variant of a spec function optimized for implementation rather than readability. More concretely, it has simpler-to-satisfy pre-conditions and/or more useful post-conditions. For instance, instead of the automatically-generated pre-conditions, we use the tidy pre-conditions we wanted to write in the spec but didn't because we didn't want to prove them sufficient. For instance, we could use

```
  requires |bits|<power2(61);
  ensures IsWordSeqOfLen(hash, 8);
```

as the signature for the premium version of SHA_B.

7.4 Systems often use bounded integer types

Dafny only supports integer types int and nat, both representing unbounded-size values. However, nearly all of our code concerns bounded-size integers such as bits, bytes, and 32-bit words. This led to many more annotations and proofs than we would have liked. We have provided this feedback to Dafny's author, who consequently plans to add refinement types.

7.5 Libraries should start with generality

Conventional software development wisdom is to start with simple, specific code and generalize only as needed, to avoid writing code paths which are not exercised or tested. We found this advice invalid in the context of verification: instead, it is often easier to write, prove, and use a more-general statement than the specific subset we actually need. For example, rather than reason about the behavior of shifting a 32-bit integer by k bits, it is better to reason about shifting n-bit integers k bits. Actual code may be limited to $n = 32$, but the predicates and lemmas are easier to prove in general terms.

7.6 Spec reviews are productive

Independent spec reviews (§3.3) caught multiple human mistakes; for instance, we caught three bugs in the segmentation spec that would have prevented our code from working the first time. Similarly, we found two bugs in the SHA-1 spec; these were easily detected, since the spec was written to closely match the text of the FIPS spec [50].

To our knowledge, only three mistakes survived the review process, and all three were liveness, not security, bugs in the TPM spec: Code written against the original spec would, under certain conditions, wait forever for an extra reply byte which the TPM would never send.

Also, our experience was consistent with prior observations that the act of formal specification, even before verification, clarifies thinking [38]. This discipline shone especially in specifying hardware interfaces, such as x86 segmentation behavior. Rather than probing the hardware's behavior with a code-test-debug cycle, specification required that we carefully extract and codify the relevant bits of Intel's Byzantine documentation. This led to a gratifying development experience in which our code worked correctly the first time we ran it.

7.7 High-level tools have bugs

One of our central tenets is that verification should be performed on the low-level code that will actually run, not the high-level code it is compiled from. This is meant to reduce bugs by removing the compiler from the TCB. We

found that this is not just a theoretical concern; we discovered *actual* bugs that this approach eliminates.

For example, when testing our code, we found a bug in the Dafny-to-C# compiler that suppressed calls to methods with only ghost return values, even if those methods had side effects. Also, we encountered a complex bug in the translation of while loops that caused the high-level Dafny verifier to report incorrect code as correct. Finally, verifying at the assembly level caught multiple bugs in DafnyCC, from errors in its variable analysis and register allocator to its handling of calculational proofs.

8 Evaluation

We claim that it is feasible to engineer apps fully verified to adhere to a security-sensitive specification. We evaluate this claim by measuring the artifacts we built and the engineering effort of building them.

8.1 System size

Table 1 breaks down the components of the system. It shows the size of the various specs, high-level implementation code, and proof statements needed to convince the verifier that the code meets the specs. It also shows the amount of verifiable assembly code, most generated by DafnyCC but some written by hand. Overall, Ironclad consists of 3,546 lines of spec, plus 7K lines of implementation that compile to 42K assembly instructions. Verifying the system takes 3 hours for functional-correctness properties and an additional 21 hours for relational properties.

Component	**Spec** (SLOC)	**Impl** (SLOC)	**Proof** (SLOC)	**Asm** (LOC)	**Boogie** time (s)	**SymDiff** time (s)
Specific apps:						
PassHash	32	81	193	447	158	6434
TrInc	78	232	653	1292	438	9938
Notary	38	140	307	663	365	14717
DiffPriv	444	586	1613	3523	891	21822
Ironclad core:						
App common	43	64	119	289	210	0
SHA-1,-256	420	574	3089	6049	698	0
RSA	492	726	4139	3377	1405	9861
BigNum	0	1606	8746	7664	2164	0
UDP/IP stack	0	135	158	968	227	4618
Seqs and ints	177	312	4669	1873	791	888
Datatypes	0	0	0	5865	1827	0
Core math	72	206	3026	476	571	0
Network card	0	336	429	2126	199	3547
TPM	296	310	531	2281	417	0
Other HW	90	324	671	2569	153	3248
Modified Verve:						
CPU/memory	900	643	2131	260	67	0
I/O	464	410	1126	1432	53	1533
GC	0	286	1603	412	92	0
Total	3546	6971	33203	41566	10726	76606

Table 1: **System Line Counts and Verification Times.** *Asm LOC includes both compiled Dafny and hand-written assembly.*

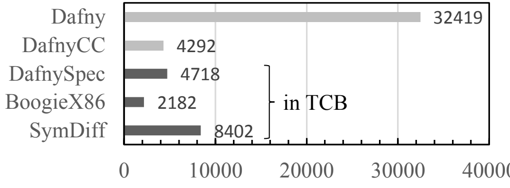


Figure 6: **Tool Line Counts.**

Most importantly, our specs are small, making manual spec review feasible. Altogether, all four apps have 3,546 SLOC of spec. The biggest spec components are hardware and crypto. Both these components are of general use, so we expect spec size to grow slowly as we add additional apps.

Our ratio of implementation to spec is 2:1, lower than we expected. One cause for this low ratio is that much of the spec is for hardware, where the measured implementation code is just drivers and the main implementation work was done by the hardware manufacturers. Another cause is that we have done little performance optimization, which typically increases this ratio.

Our ratio of proof annotation to implementation, 4.8:1, compares favorably to seL4's ∼20:1. We attribute this to our use of automated verification to reduce the burden on developers. Note also that the ratio varies across components. For instance, the core system and math libraries required many proofs to establish basic facts (§7.2); thanks to this work, higher-level components obtained lower ratios. Since these libraries are reusable, we expect the ratio to go down further as more apps reuse them.

Figure 6 shows line counts for our tools. The ones in our TCB have 15,302 SLOC. This is much less than the 32,419 SLOC in the original Dafny-to-C# compiler, let alone the code for the C# compiler. Our DafnyCC tool is 4,292 SLOC and depends on Dafny as well, but as discussed earlier (§3.4), it is not in the TCB.

8.2 Developer effort

Previous work based on interactive proof assistants showed that the costs can be quite high [48]. In contrast, our experience suggests that automated provers reduce the burden to a potentially tolerable level. Despite learning and creating new tools, as well as several major code refactorings, we constructed the entire Ironclad system with under three person-years of effort.

8.3 Performance

Finally, although performance was not one of our goals, we evaluate the performance of our apps to demonstrate how much more work lies ahead in optimization. For these experiments, we use as our server an HP Compaq 6005 Pro PC with a 3-GHz AMD Phenom II X3 CPU, 4 GB of RAM, and a Broadcom NetXtreme Gigabit Eth-

Operation	Dominant step	Ironclad		Unverified		Slowdown
Notary notarize	Compute RSA signature	934 ms	±0 ms	8.88 ms	±0.68 ms	105
TrInc create	Compute reciprocal of RSA modulus	4.96 ms	±0.03 ms	866 μs	±507 μs	5.72
TrInc advance	Compute RSA signature	1.01 s	±0.00 s	12.1 ms	±0.2 ms	83.6
PassHash hash	Compute SHA-256 hash	276 μs	±10 μs	159 μs	±3 μs	1.73
DiffPriv initialize_db	None, just network overhead	168 μs	±2 μs	155 μs	±3 μs	1.08
DiffPriv add_row	Decrypt RSA-encrypted row	944 ms	±17 ms	8.85 ms	±0.06 ms	107
DiffPriv query	Compute noise with BigInts	126 ms	±19 ms	668 μs	±36 μs	189

Table 3: **App benchmarks.** *Latency of different request types, as seen by a client on the same network switch, for Ironclad Apps and unverified variants written in C#. Ranges shown are 95% confidence intervals for the mean.*

Op	Param	Ironclad	OpenSSL	C#/.NET
TPM GetRandom	256b	39 μs/B	*–	*–
RSA KeyGen	1024b	39.0 s	*12 ms	*181 ms
RSA Public	1024b	35.7 ms	204 μs	65 μs
RSA Private	1024b	858 ms	4.03 ms	1.40 ms
SHA-256	256B	26 ns/B	13 ns/B	24 ns/B
SHA-256	8192B	13 ns/B	10 ns/B	7.6 ns/B

Table 2: **Crypto microbenchmarks.** **Only Ironclad uses the TPM for KeyGen.*

ernet NIC. As our client, we use a Dell Precision T7610, with a 6-core 2.6-GHz Xeon E5-2630 CPU, 32 GB of RAM, and an Intel 82579LM Gigabit Ethernet NIC. The two are connected to the same gigabit switch.

Table 2 shows the latency and bandwidth of various low-level crypto and TPM operations; these operations constitute the dominant cost of app initialization and/or app operations. Table 2 also shows the corresponding times for C#/.NET code on Windows 7 and OpenSSL on Ubuntu. The results show that our RSA library is about two orders of magnitude slower than unverified variants, and our SHA-256 code approaches within 30%.

This poor performance is not fundamental to our approach. Indeed, verification provides a safety net for aggressive optimizations. For example, we extended DafnyCC to support directed inlining, applied this to the code for SHA-256, then performed limited manual optimization on the resulting verifiable assembly. This more than doubled our code's performance, bringing us within 30% of OpenSSL. Along the way, we had no fear of violating correctness; indeed, the verifier caught several bugs, e.g., clobbering a live register. We used the same technique to manually create a verified assembly-level unrolled add function for BigIntegers. Similarly, verification helped to correctly move our multi-precision integer library from immutable sequences to mutable arrays, making it 1000× faster than the first version. Many optimization opportunities remain, such as unrolled loops, inlined procedures, and arithmetic using the Chinese remainder theorem and Montgomery form.

Next, we show the performance of high-level operations. To compare Ironclad's performance to unverified servers, we wrote unverified variants of our apps in C# using the .NET Framework. We run those apps on the server on Windows 7.

Table 3 shows the results. We measure various operations from the client's perspective, counting the time between sending a request and receiving the app's reply. For each operation, we discard the first five results and report the mean of the remaining 100 results; we also report the 95% confidence interval for this mean. We use 1,024-bit RSA keys, 32-byte hashes for `notarize` and `advance`, 12-byte passwords and 16-byte salts for `hash`, 20-byte nonces and four-column rows for `add_row`, and a 19-instruction mapper for `query`.

The results are generally consistent with the microbenchmark results. Slowdowns are significant for operations whose dominant component involves an RSA key operation (`notarize`, `advance`, `add_row`), and lower but still substantial for those involving SHA-256 (`hash`) and big-integer operations (`create` and `query`). The `initialize_db` operation, which involves no cryptographic operations and essentially just involves network communication, incurs little slowdown.

9 Limitations and Future Work

As with any verified system, our guarantees only hold if our specs, both of hardware and apps, are correct. While we strive to keep the specs minimal and take additional steps to add assurance (§3.3), this is the most likely route for errors to enter the system.

We also rely on the correctness of our verification tools, namely our Dafny spec translator, SymDiff, Boogie, and Z3. Unfortunately, these tools do not currently provide proof objects that can be checked by a small verifier, so they all reside in our TCB. Fortunately, our spec translator is tiny, and Boogie and Z3 are extensively tested and used by dozens of projects, including in production systems. In practice, we did not encounter any soundness bugs in these tools, unlike the untrusted, higher-level tools we employed (§7.7).

At present, we do not model the hardware in enough detail to prove the absence of covert or side channels that may exist due to timing or cache effects, but prior work [48] suggests that such verification is feasible.

Currently, we prove the functional correctness and noninterference of our system, but our proofs could be extended in two directions that constitute ongoing work: proving liveness, and connecting our guarantees to even higher-level cryptographic protocol correctness proofs. For example, we want to explicitly reason about probability distributions to show that our use of cryptographic primitives creates a secure channel [6, 9].

With Ironclad, we chose to directly verify all of our code rather than employing verified sandboxing. However, our implementation supports provably correct page table usage and can safely run .NET code, so future work could use unverified code as a subroutine, checking its outputs for desired properties. Indeed, type safety allows code to safely run in kernel mode, to reduce kernel-user mode crossings.

10 Related Work

Trusted Computing. As discussed in §1, Trusted Computing has produced considerable research showing how to identify code executing on a remote machine [53]. However, with a few exceptions, it provides little guidance as to how to assess the security of that code. Property-based attestation [58] shifts the problem of deciding if the code is trustworthy from the client to a trusted third party, while semantic attestation attests to a large software stack—a traditional OS and managed runtime—to show that an app is type safe [29]. The Nexus OS [60] attests to an unverified kernel, which then provides higher-level attestations about the apps it runs. In general, verification efforts in the Trusted Computing space have focused primarily on the TPM's protocols [11, 16, 27, 44] rather than on the code the TPM attests to.

Early security kernels. Issued in 1983, the DoD's "Orange Book" [18] explicitly acknowledged the limitations of contemporary verification tools. The highest rating (A1) required a formal specification of the system but only an informal argument relating the code to the specification. Early efforts to attain an A1 rating met with mixed success; the KVM/370 project [25] aimed for A1, but, due in part to inadequacies of the languages and tools available, settled for C2. The VAX VMM [34] did attain an A1 rating but could not verify that their implementation satisfied the spec. Similar caution applies to other A1 OSes [22, 59].

Recent verified kernels. The seL4 project [35, 36, 48] successfully verified a realistic microkernel for strong correctness properties. Doing so required roughly 200,000 lines of manual proof script to verify 8,700 lines of C code using interactive theorem proving (and 22 person-years); Ironclad's use of automated theorem proving reduces this manual annotation overhead, which helped to reduce the effort required (3 person-years). seL4 has focused mainly on kernel verification; Ironclad contains the small Verve verified OS, but focuses more on library (e.g. BigNum/RSA), driver (e.g. TPM), and application verification in order to provide whole-system verification. seL4's kernel is verified, but can still run unverified code outside kernel mode. Ironclad currently consists entirely of verified code, but it can also run unverified code (§9). Both seL4 and Ironclad verify information flow.

Recent work by Dam et al. [14] verifies information-flow security in a simple ARM separation kernel, but the focus is on providing a strict separation of kernel usages among different security domains. This leaves other useful security properties, including functional correctness of the OS and applications, unverified.

While seL4 and Ironclad Apps run on commodity hardware, the Verisoft project [2] aimed for greater integration between hardware and software verification, building on a custom processor. Like seL4, Verisoft required >20 person-years of effort to develop verified software.

Differential privacy. Many systems implement differential privacy, but none provide end-to-end guarantees about their implementations' correctness. For instance, Barthe et al. describe *Certipriv* [8], a framework for mechanically proving algorithms differentially private, but do not provide an executable implementation of these algorithms. As a consequence, implementations have vulnerabilities; e.g., Mironov [46] demonstrated an attack that affected PINQ [43], Airavat [56], Fuzz [55], and GUPT [47].

11 Conclusion

By using automated tools, we have verified full-system, low-level, end-to-end security guarantees about Ironclad Apps. These security guarantees include non-trivial properties like differential privacy, which is notoriously difficult to get right. By writing a compiler from Dafny to verified assembly language, we verified a large suite of libraries and applications while keeping our tool and specification TCB small. The resulting system, with ∼6500 lines of runnable implementation code, took ∼3 person-years to verify. Beyond small, security-critical apps like Ironclad, verification remains challenging: assuming ∼2000 verified LOC per person-year, a fully verified million-LOC project would still require ∼100s of person-years. Fortunately, the tools will only get better, so we expect to see full-system verification scale to larger systems and higher-level properties in the years to come.

Acknowledgments

We thank Jeremy Elson, Cedric Fournet, Shuvendu Lahiri, Rustan Leino, Nikhil Swamy, Valentin Wuestholz, Santiago Zanella Beguelin, and the anonymous reviewers for their generous help, guidance, and feedback. We are especially grateful to our shepherd Gernot Heiser, whose insightful feedback improved the paper.

References

[1] Advanced Micro Devices. AMD64 Architecture Programmer's Manual. AMD Publication no. 24593 rev. 3.22, 2012.

[2] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. *Automated Reasoning*, 42(2–4):389–454, 2009.

[3] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, and G. Proudler. *Trusted Computing Platforms – TCPA Technology in Context*. Prentice Hall, 2003.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *Proceedings of Formal Methods for Components and Objects (FMCO)*, 2006.

[5] G. Barthe, C. Fournet, B. Gregoire, P.-Y. Strub, N. Swamy, and S. Z. Beguelin. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, Jan. 2014.

[6] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of IACR CRYPTO*, 2011.

[7] G. Barthe, B. Grégoire, Y. Lakhnech, and S. Zanella-Béguelin. Beyond provable security. Verifiable IND-CCA security of OAEP. In *Proceedings of CT-RSA*, 2011.

[8] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2012.

[9] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[10] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2011.

[11] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Replay attack in TCG specification and solution. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2005.

[12] C. Cachin. Distributing trust on the Internet. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2001.

[13] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics*, 2009.

[14] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[15] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2008.

[16] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[17] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[18] Department of Defense. *Trusted Computer System Evaluation Criteria*. National Computer Security Center, 1983.

[19] C. Dwork. Differential privacy. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2006.

[20] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the IACR Theory of Cryptography Conference (TCC)*. 2006.

[21] R. W. Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*, 1967.

[22] L. J. Fraim. SCOMP: A solution to the multilevel security problem. *Computer*, 16:26–34, July 1983.

[23] A. Ghosh, T. Roughgarden, and M. Sundararajan. Universally utility-maximizing privacy mechanisms. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 2009.

[24] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1982.

[25] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1984.

[26] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2009.

[27] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga. Security evaluation of scenarios based on the TCG's TPM specification. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2007.

[28] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3, 1991.

[29] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *Proceedings of the Conference on Virtual Machine Research*, 2004.

[30] G. Heiser, L. Ryzhyk, M. von Tessin, and A. Budzynowski. What if you could actually trust your kernel? In *Hot Topics in Operating Systems (HotOS)*, 2011.

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[32] Intel Corporation. Intel Trusted Execution Technology – Measured Launched Environment Developer's Guide. Document number 315168-005, June 2008.

[33] B. Kaliski. PKCS #1: RSA cryptography specifications version 1.5. RFC 2313, Mar. 1998.

[34] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, Nov. 1991.

[35] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), 2014.

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[37] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of Computer Aided Verification (CAV)*, July 2012.

[38] L. Lamport. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[39] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[40] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[41] M. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. Kelley, R. Shay, and B. Ur. Measuring password guessability for an entire university. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[42] F. Mckeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[43] F. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2009.

[44] J. Millen, J. Guttman, J. Ramsdell, J. Sheehy, and B. Sniffen. Analysis of a measured launch. Technical Report 07-0843, The MITRE Corporation, June 2007.

[45] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3), 1976.

[46] I. Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[47] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. E. Culler. GUPT: Privacy preserving data analysis

made easy. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.

[48] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[49] National Institute of Standards and Technology. The keyed-hash message authentication code (HMAC), 2012. FIPS PUB 198-1.

[50] National Institute of Standards and Technology. Secure hash standard (SHS), 2012. FIPS PUB 180-4.

[51] National Vulnerability Database. Heartbleed bug. CVE-2014-0160 http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160, Apr. 2014.

[52] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[53] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[54] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1), 1980.

[55] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the ACM International Conference on Functional Programming*, 2010.

[56] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[57] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[58] A.-R. Sadeghi and C. Stueble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *Proceedings of the Workshop on New Security Paradigms (NSPW)*, 2004.

[59] W. Shockley, T. Tao, and M. Thompson. An overview of the GEMSOS class A1 technology and application experience. In *Proceedings of the National Computer Security Conference*, Oct. 1988.

[60] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[61] G. Smith. Principles of secure information flow analysis. *Malware Detection*, pages 297–307, 2007.

[62] S. G. Stubblebine and P. F. Syverson. Fair on-line auctions without special trusted parties. In *Proceedings of Financial Cryptography*, 1999.

[63] Trusted Computing Group. Trusted Platform Module Main Specification. Version 1.2, Revision 116, 2011.

[64] Wave Systems Corp. Trusted Computing: An already deployed, cost effective, ISO standard, highly secure solution for improving Cybersecurity. http://www.nist.gov/itl/upload/Wave-Systems_Cybersecurity-NOI-Comments_9-13-10.pdf, 2010.

[65] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[66] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

SHILL: A Secure Shell Scripting Language

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong
Harvard School of Engineering and Applied Sciences

Abstract

The Principle of Least Privilege suggests that software should be executed with no more authority than it requires to accomplish its task. Current security tools make it difficult to apply this principle: they either require significant modifications to applications or do not facilitate reasoning about combining untrustworthy components.

We propose SHILL, a secure shell scripting language. SHILL scripts enable compositional reasoning about security through contracts that limit the effects of script execution, including the effects of programs invoked by the script. SHILL contracts are declarative security policies that act as documentation for consumers of SHILL scripts, and are enforced through a combination of language design and sandboxing.

We have implemented a prototype of SHILL for FreeBSD and used it for several case studies including a grading script and a script to download, compile, and install software. Our experience indicates that SHILL is a practical and useful system security tool, and can provide fine-grained security guarantees.

1 Introduction

Users of commodity operating systems often need to execute untrustworthy software. In fact, this is the common case: due to errors or malicious intent, software regularly does not behave as expected. The Principle of Least Privilege (POLP) [31] requires that software should be given only the authority it needs to accomplish its functionality. If adhered to, this principle (also known as the Principle of Least Authority) can help protect systems from erroneous or malicious software.

However, commodity systems and their secure tools fail to adequately support POLP. First, it is difficult for the user of a commodity system to determine what authority a given piece of software requires to execute correctly. Second, current mechanisms for limiting authority are difficult to use: they are either coarse-grained or require significant changes to existing software, and are often not available to all users [16]. For both of these reasons, users tend to execute software with more authority than is necessary.

For example, consider scripts to grade homework submissions in a computer science course. Students submit source code, and a script `grade.sh` is run on each submission to compile it and run it against a test suite. The submission server must execute `grade.sh` with sufficient authority to accomplish its task, but should also restrict its authority to protect the server from student-submitted code and ensure the integrity of grading. At a coarse grain, the server should allow `grade.sh` to access files and directories necessary to compile, run, and record the scores of homework submissions, and deny access to other files or resources. This ensures, for example, that a careless student's code won't corrupt the server and a cheating student's code won't modify or leak the test suite. At a fine grain, each call to `grade.sh` to grade a single submission should be isolated from the grading of other submissions. This ensures, for example, that a cheating student cannot copy solutions from another submission.

Securing a script such as `grade.sh` is difficult, as it requires balancing functional and security requirements. To begin with, it is a priori unclear what authority `grade.sh` needs to execute correctly. While the author of the script may know, the user must examine the code to try to determine what authority it requires. If the user can identify the required resources, she can use existing tools for sandboxing program execution (e.g., [20, 3, 15, 14]) to achieve the coarse-grained security requirements. However, it is difficult to use the same tools to enforce the fine-grained security requirements described above. This is because achieving these requirements requires that each invocation of `grade.sh` is given different privileges, i.e., it must be executed in a differently configured sandbox. Configuring all of these sandboxes correctly is error prone, so users often forgo

```
provide grade :
  {submission : is_file && readonly,
   tests : is_dir && readonly,
   working : dir(+create_dir with full_priv),
   grade_log : is_file && writeable,
   wallet : ocaml_wallet} → void;
```

Figure 1: SHILL contract for a grading script

fine-grained security and violate POLP.

To address these issues, we introduce the SHILL programming language. SHILL is a secure shell scripting language with features that help apply POLP in commodity operating systems.[1] At the core of SHILL are declarative security policies that describe and limit the effects of script execution, including effects of arbitrary programs invoked by the script.

These declarative security policies can be used by producers of software to provide fine-grained descriptions of the authority the software needs to execute. This, in turn, allows consumers of software to inspect the software's required authority, and make an informed decision to execute the software, reject the software, or apply a more restrictive policy on the software. The SHILL runtime system ensures that script execution adheres to the declared security policy, providing a simple mechanism to restrict the authority of software.

Two key features enable SHILL's declarative security policies: language-level *capabilities* and *contracts*. SHILL scripts access system resources only through capabilities: unforgeable tokens that confer privileges on resources. SHILL scripts receive capabilities only from the script invoker; SHILL scripts cannot store or arbitrarily create capabilities. Moreover, SHILL uses *capability-based sandboxes* to control the execution of arbitrary software. Thus, the capabilities that a user passes to a SHILL script limit the script's authority, including any programs it invokes. SHILL's contracts specify what capabilities a script requires and how it intends to use them. SHILL's runtime and sandboxes enforce these contracts, hence they serve as fine-grained, expressive, declarative security policies that bound the effects of a script.

For example, Figure 1 shows a SHILL contract for a script to grade a single student submission (corresponding to the `grade.sh` script described above). It is a declarative security specification for the function grade, which takes 5 arguments: a read-only file submission (i.e., the student's source code), a read-only directory tests (containing the test suite), a "working directory" in which the script may create subdirectories with full privileges, a writeable file grade_log for recording the student's grade, and a "wallet" that provides sufficient capabilities to invoke the OCaml compiler. This contract serves two purposes: it clearly describes what grade needs to execute correctly and it also provides guarantees about what grade may do when invoked. Given this contract, a user can be confident that grade satisfies the security requirements described above, even though grade will compile and execute student-submitted code. Specifically: grade will not read any other student's submission; grade will not communicate over the network (as it has no capability for network access); grade will not corrupt the test suite nor write any files other than the grade log and subdirectories it creates within the working directory. The implementation of grade (not shown) focuses solely on the functionality for grading, and is not concerned with enforcing security requirements.

SHILL offers language abstractions for reasoning about the authority of pieces of software and their composition. Specifically, SHILL (1) introduces a capability-based scripting language with language abstractions (such as contracts and wallets) to use capabilities effectively, and (2) implements, on a commodity operating system, capability-based sandboxes that extend the guarantees of the scripting language to binary executables and legacy applications. These language abstractions, and the enforcement of these abstractions, make it possible to manage authority and follow POLP, even when using and combining untrusted programs.

The rest of the paper is structured as follows. In Section 2 we present the design of SHILL. Our implementation of SHILL in FreeBSD 9.2 is described in Section 3. We evaluate SHILL by using it to implement several case studies, and measure the overhead of SHILL's security mechanisms. We present the evaluation results in Section 4. Section 5 describes related work.

2 Design and security of SHILL

SHILL aims to meet the following five goals:

1. Script users can control the authority of a script, i.e., what system resources it can access or modify.
2. Script users can understand what authority a script needs in order to accomplish its functionality.
3. Security guarantees of scripts apply transitively to other programs the script may invoke, including arbitrary executables.
4. SHILL separates the security aspects of scripts from functional aspects, reducing the impact of security concerns on the effort required to write scripts.
5. SHILL is compatible with commodity operating system abstractions.

[1] SHILL is not an interactive shell, but rather a language that presents operating system abstractions to the programmer and is used primarily to launch programs. Other languages currently used for this purpose include Perl, Python, and the scripting portion of Bash.

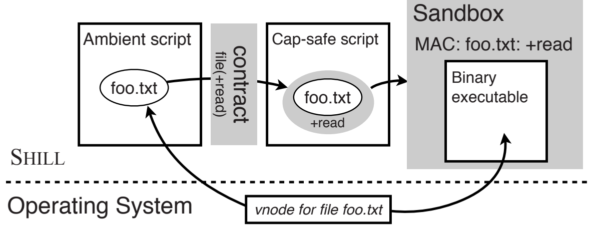


Figure 2: SHILL in a nutshell

To meet these goals, SHILL uses a combination of language design and mandatory access control-based sandboxing.

In most scripting languages, scripts can access a resource (such as a file) using the resource's well-known global name. Access control is based on the user on whose behalf the script executes. Thus, a script's authority is *ambient* (i.e., it derives from the script's execution context) [25], and a script may access any and all resources that the invoking user may access. SHILL's security is based on capabilities instead of ambient authority.

There are two kinds of SHILL scripts: *capability-safe* SHILL *scripts*, and *ambient* SHILL *scripts*. Capability-safe SHILL scripts play the same role as regular shell scripts, but do not have ambient authority and must be given capabilities to access resources. Ambient SHILL scripts are used to create the initial set of capabilities to give to capability-safe scripts. They do have ambient authority, but are very restricted: ambient scripts can only create capabilities for system resources and invoke capability-safe SHILL scripts.

Each capability-safe SHILL script comes with a contract that is enforced by the language runtime. A capability-safe SHILL script can use the capabilities it possesses to access resources using SHILL's built-in functions, if allowed by the contract. SHILL scripts can also invoke arbitrary executables in *capability-based sandboxes*. A capability-based sandbox is created with a set of capabilities, and enforces a mandatory access control policy that restricts the executable's behavior based on those capabilities and their contracts.

Figure 2 depicts the life cycle of a capability for a file named foo.txt. First, an ambient script acquires a capability for the file from the operating system using the user's ambient authority. This capability is then passed to a capability-safe script via a contract, which restricts the privileges on the capability to +read (i.e., the capability can be used only to read foo.txt, not to write to it, etc.). The capability-safe script then runs an executable in a sandbox, granting it the capability to read the file.

Threat model In SHILL's threat model, some capability-safe scripts (and the executables they invoke) are not trusted. However, their behavior is restricted by their contracts and the capabilities they are given: a capability-safe script (and any executables it invokes) can access resources only as permitted by its contract and the capabilities it possesses. Of course, the contract that accompanies a script may also be untrustworthy: a user should inspect the contract and understand its security implications before passing capabilities to the script. The benefit of SHILL's approach is that it is much easier to inspect and understand the declarative contract than to examine the script itself.

SHILL's trusted computing base includes the operating system kernel and SHILL runtime. SHILL does not explicitly defend against malicious scripts or executables that exploit security flaws in the kernel or SHILL itself.

The rest of this section describes how SHILL's design and features contribute towards these goals, and provides an introduction to SHILL via several small examples.

2.1 Controlling script authority

Ambient authority makes writing scripts easy: if a script needs to access a resource, it can simply use the resource's name to access it. However, ambient authority makes it difficult to understand and control the potential effect of executing a script. First, the authority of a script is not easily deducible from its code, a problem that is exacerbated when the script invokes other scripts or executables. Second, commodity operating systems do not provide easy mechanisms to limit authority of an execution context, for example, by allowing a user to temporarily restrict permissions in a fine-grained way.

Authority in SHILL is controlled by *capabilities*. In order to access a resource, a SHILL script must have a capability for that resource. SHILL scripts can only acquire capabilities as arguments provided by the user, or by deriving them from other capabilities (e.g., using a directory capability to acquire a capability for a file in the directory). These restrictions, known as *capability safety*, lie at the heart of the security of SHILL scripts. Capability safety makes it possible for users to control the authority of SHILL scripts they invoke (Goal 1).

Figure 3 presents a snippet of SHILL code that demonstrates how SHILL scripts use capabilities. It defines a function find_jpg for recursively finding all the files with extension .jpg within a given directory. Argument cur is a capability for either a file or a directory. In contrast with standard scripting languages, cur is not a string that names a file, but is a capability that denotes it, much like a file descriptor. If cur is a file capability and the name of the file ends with .jpg, then the script uses the built-in function path to get the string for the path to the file,[2] and

[2]The library function has_ext also uses path.

```
1  find_jpg = fun(cur,out) {
2    # if cur is a file with extension jpg,
3    # output its path to out.
4    if is_file(cur) && has_ext(cur, "jpg") then
5      append(out, path(cur));
6
7    # if cur is a directory, recur on its contents
8    if is_dir(cur) then
9      for name in contents(cur) {
10       child = lookup(cur, name);
11       if !is_syserror(child) then
12         find_jpg(child, out);
13     }
14 }
```

Figure 3: SHILL script snippet to find .jpg files

appends it to the pipe or file capability out (lines 4–5).

If cur is a directory capability, then the built-in function contents is used to get the list of names of children of cur. For each child, the script calls lookup(cur, name) to obtain a capability for the child (line 10), which is then used in a recursive call to find_jpg (line 12).

Conceptually, SHILL capabilities correspond to operating system representations of resources, such as file descriptors, and built-in functions such as append and lookup are wrappers for the corresponding system calls.

SHILL enforces capability safety by restricting the expressiveness of the scripting language. While SHILL offers full-fledged language features and rich libraries, comparable to other scripting languages, the built-in functions for using resources require capabilities as arguments. In addition, SHILL does not have mutable variables and capabilities are not serializable. This means that SHILL scripts cannot store or share capabilities through memory, the filesystem, or the network. For controlled sharing of capabilities, SHILL provides *wallets*, capabilities for packaging and managing collections of capabilities. We discuss wallets further in Section 2.4.1.

SHILL scripts provide the same protection from confused deputy attacks [12] as traditional capability systems. Furthermore, filesystem operations that produce new capabilities (such as lookup) do not allow scripts to arbitrarily traverse the filesystem. For instance, a script cannot use the capability for the current directory cur and lookup(cur,"..") to obtain the parent directory of cur.

2.2 Contracts

Capability safety makes it possible to limit the authority granted to a SHILL script by carefully selecting what capabilities to pass as arguments. Unfortunately, needing to pass capabilities explicitly makes it harder for script users to deduce how to use scripts and compose them to complete more complicated tasks. At its core, this is a problem of defining the script's interface: how does the script communicate what resources it requires and how it will use those resources?[3]

SHILL addresses these issues by providing expressive, fine-grained and enforceable interfaces for scripts (Goal 2) following the *Design by Contract* paradigm [23, 24]. Every function that a SHILL script exports (i.e., makes available to users of the script) is accompanied by a *contract* that describes the arguments the function expects and the result it returns. For example, the following snippet is a contract for the find_jpg function from Figure 3:

```
provide find_jpg :
  {cur : is_dir ∨ is_file, out : is_file} → void;
```

The **provide** keyword indicates that the function find_jpg is exported. The contract for the function is {cur : is_dir ∨ is_file, out : is_file} → void. Each function contract has two parts: the precondition and the postcondition. The precondition of our example states that find_jpg takes two arguments: a capability cur that is either a directory or a file capability, and a file capability out. Following Unix convention, file capabilities include capabilities for files, pipes, and devices. The postcondition void means that no value is returned.

The precondition of the contract above describes what kind of capabilities find_jpg needs, but does not indicate how the function intends to use these capabilities. SHILL allows us to give a more precise contract for find_jpg:

```
provide find_jpg :
  {cur : dir(+contents, +lookup, +path) ∨ file(+path),
   out : file(+append)} → void;
```

This version specifies not only what kind of capabilities the function consumes but also what privileges it requires on these capabilities. Each privilege, such as +path or +contents, corresponds to an operation on a capability. A capability contract with a set of privileges restricts what operations that capability can be used for.

Some operations on capabilities, such as lookup, produce more capabilities. Capability contracts can specify the privileges a script should have on these derived capabilities. For example, privilege +lookup with { +path, +stat } indicates that any capabilities derived using the lookup operation should only have the +path and +stat privileges. When a privilege confers the right to derive new capabilities but does not come with a modifier (such as the +lookup privilege in the contract for find_jpg), the derived capability has the same privileges as its parent capability.

Each contract establishes an agreement between two

[3]Traditional shell scripting languages such as Bash or Python also suffer from these issues, but the use of ambient authority masks them: scripts typically receive much more authority than needed.

parties: the provider of the value with the contract and the value's consumer. As part of the agreement, each party promises to live up to its contractual obligations. In this way, a contract both describes a guarantee one party provides and a requirement the other party demands. For function contracts, the consumer's obligations are to supply function arguments that satisfy the precondition, and the provider must produce a result that satisfies the postcondition. For capability contracts, the provider agrees to provide a capability of the appropriate kind with *at least* the specified privileges while the consumer promises to use the capability as if it has *at most* the specified privileges. For example, according to the find_jpg contract, users of find_jpg must supply a file capability that permits the append operation for the out argument, while find_jpg itself promises not to call other operations on the capability, such as read.

The SHILL runtime checks whether parties live up to their obligations by monitoring execution and checking that values are used in accordance with their contracts. For example, when find_jpg is called with a capability for a directory and a capability for the output file, the body of find_jpg does not receive the capabilities themselves. Instead, each contract wraps the underlying capability with a *proxy*. These proxies enforces the contracts for cur and out by intercepting calls to operations on the capabilities and allow them only if permitted by the contract. If the body of find_jpg attempts to perform an operation that isn't permitted—such as reading the contents of out or unlinking cur—the proxy will indicate that a contract violation has occurred. If a contract is violated, the SHILL runtime aborts execution and, to help with auditing and debugging, indicates which part of the script failed to meet its obligations.

2.3 Securing arbitrary executables

SHILL security guarantees must be completely enforced: even if a script calls other scripts or runs arbitrary executables, its authority should be restricted to its capabilities, and it should meet its contract obligations (Goal 3). When SHILL scripts invoke only other SHILL scripts, we achieve SHILL's security guarantees easily because of the language's semantics. However, scripts also invoke executable programs.

To ensure that these programs cannot violate SHILL's security guarantees, SHILL scripts may only invoke executables inside a *capability-based sandbox*. When a sandbox is created, it is given a set of capabilities. The SHILL sandbox limits the authority of the sandboxed executable to the authority implied by the set of capabilities.

Scripts can invoke an executable in a sandbox by calling the built-in function exec. For example, the following snippet executes the file jpeginfo in a sandbox with the arguments -i and a given file:

```
exec(jpeginfo, ["jpeginfo","-i",file], stdout = out,
     extras = [libc,libjpeg])
```

The exec function has two required arguments. The first is a file capability with the +exec privilege. The second is a list of string arguments to provide to the executable. SHILL programmers can also provide as arguments to executables capabilities for files or directories instead of string representations of their paths. In this case, the path to the given file is passed to the executable as an argument. The exec function also takes some optional arguments, including capabilities to use for standard input, output, or error (stdout = out), and extra capabilities needed by the program (extras = [libc,libjpeg]). This set of extra capabilities is often quite large. In Section 2.4.1, we describe abstractions to help manage capabilities for sandboxes.

SHILL sandboxes enforce a capability-based mandatory access control (MAC) policy on the sandboxed execution. For example, the sandbox for jpeginfo allows access only to resources indicated by capabilities passed as arguments to exec (which, for the jpeginfo example above, are the jpeginfo, file, out, libc, and libjpeg files and directories). Moreover, if any of these capabilities comes with a contract, the MAC policy further limits access to the resource according to the capability's contract.

This capability-based MAC policy is enforced *in addition* to the operating system's discretionary access control (DAC) policies: an operation on a resource by a sandboxed execution is permitted only if it passes the checks performed by the operating system based on the user's ambient authority and is also permitted by the capabilities possessed by the sandbox. Note that sandboxed executables never possess capabilities that allow them to circumvent the MAC policy. For example, no sandboxed executable has a capability to unload kernel modules, including the module that enforces the MAC policy. Section 3.2 describes how we implement capability-based sandboxes using the TrustedBSD MAC framework.

2.4 Writing SHILL scripts

SHILL's security benefits come at the cost of extra effort to write scripts. Nonetheless, we strive to make it easy to write SHILL scripts while obtaining stronger security guarantees than traditional shell scripting languages. To make it easier to write scripts, SHILL offers security abstractions such as *capability wallets* and pushes security concerns to the interfaces between scripts.

2.4.1 Security abstractions

SHILL requires that any access of a protected resource requires an appropriate capability. However, even sim-

```
1 provide jpeginfo :
2   {wallet : native_wallet, out : file(+write,+append),
3    arg : file(+read,+path)} → void;
4
5 jpeginfo = fun (wallet,out,arg) {
6   jpeg_wrapper = pkg_native("jpeginfo",wallet);
7   jpeg_wrapper(["-i",arg],stdout = out);
8 }
```

Figure 4: Executing jpeginfo in a sandbox using wallets

ple executable programs require access to a surprising number of files. For example, executing `cat` in a sandbox requires providing eight capabilities to libraries and configuration files in addition to capabilities for the executable itself and the input and output.

Consider a SHILL script that executes `cat` in a sandbox. One can imagine a contract that requires a separate argument for each of the eight capabilities that `cat` requires. While precise, such a contract imposes a significant burden on both the script writer (since the need for these capabilities will be exposed in the interface for the script) and the script user (who will need to supply these capabilities individually).

Another possibility is a contract that takes important capabilities separately (e.g., for the executable and the input and output) and takes all other capabilities in a list. Although succinct, this contract burdens the script's user, who has no idea what capabilities should be in this list.

We introduce *capability wallets* as a mechanism to automate and simplify the discovery, packaging, and management of capabilities that sandboxes need to run executables. Conceptually, a capability wallet is a map from strings to lists of capabilities. To reduce the burden on script writers, SHILL provides *wallet contracts*, which describe contracts for the capabilities associated with individual keys or groups of keys. To reduce the burden on script users, SHILL provides library functions to automate the collection and packaging of capabilities into wallets.

Figure 4 shows a script that uses a capability wallet to create a sandbox for the program `jpeginfo`. The first argument to the jpeginfo function has the contract native_wallet (line 2). A native_wallet is a particular kind of capability wallet that can be built using functions from SHILL's standard library. It collects together the capabilities needed to invoke executables and can be used with other functions from the SHILL standard library that present a familiar path-based interface for identifying and running executables. The capabilities in a wallet are derived from capabilities the user explicitly grants to the script. Thus despite its path-based interface, a native wallet is still capability safe.

This script uses one of the standard library functions, pkg_native, to create a wrapper containing all of the capabilities needed to run the `jpeginfo` executable in a sandbox (line 6). The script then calls the wrapper, supplying the executable arguments and input and output capabilities (line 7).

SHILL's standard library comes with a rich collection of functions that construct and manipulate wallets, wallet contracts and wallet-derived sandboxes. Section 3.1.4 presents these utilities in further detail.

2.4.2 Pushing security to interfaces

SHILL's contracts allow the programmer to separate the security specification of a script from the implementation of its functionality (Goal 4). The SHILL runtime ensures that contracts are enforced, removing the need for defensive code that checks and protects the use of capabilities. Consider the find_jpg function from Figure 3: the implementation is simple, and the security guarantee is provided by its contract. This separation makes it possible to strengthen or relax a script's security guarantees by modifying its contract. Indeed, in Section 2.2 we saw two different contracts for the find_jpg function, one of which provides a more precise security guarantee.

SHILL's contract system is rich and expressive, allowing precise specifications of security guarantees. For example, users can define their own contracts by creating contract combinators and user-defined predicates written in SHILL itself.

SHILL's contracts can also be used to write security specifications that provide different guarantees to different script users. Consider the script in Figure 5. This script recursively finds files and performs an action on these files. (It is more general than the find_jpg script of Figure 3.) The function find takes three arguments: a file or directory capability cur, a function filter that is used to select files, and a function cmd to apply to all selected files. Lines 5–16 implement find's functionality. Note that this code is straightforward, and does not directly address security concerns.

Lines 1–3 define the contract for find, using a *bounded parametric-polymorphic contract*. The polymorphic contract declares that for any contract X, the function find can be called with arguments cur, filter, and cmd such that cur satisfies contract X, filter satisfies contract X → is_bool (i.e., filter is a function that expects a value that satisfies X and returns a boolean), and cmd satisfies contract X → void (i.e., cmd is a function that expects a value that satisfies X and returns no value).

The polymorphic contract is *bounded* because the contract X on capability cur that the caller provides must have at least the privileges +lookup and +contents. Moreover, the contract requires that find can use only the +lookup and +contents privileges of the cur argument or

```
1  provide find :
2  forall X with {+lookup,+contents} .
3    {cur : X, filter : X → is_bool, cmd : X → void} → void;
4
5  find = fun(cur, filter, cmd) {
6     if is_file(cur) && filter(cur) then
7        cmd(cur);
8
9     # if cur is a directory, recur on its contents
10    if is_dir(cur) then
11       for name in contents(cur) {
12          child = lookup(cur, name);
13          if !is_syserror(child) then
14             find(child, filter, cmd);
15       }
16 }
```

Figure 5: A find script with a polymorphic contract

derived capabilities, even though contract X may specify more privileges. Importantly, the contracts for arguments filter and cmd allow these functions to use all of the privileges that X specifies. In essence, the contract of find dynamically seals [28] the argument cur as it flows into the body of the function through contract X, and unseals it as it flows out to the functions filter and cmd.

The contract on find allows clients to use find in different ways. For example, one client may use it with a filter that examines file creation times (which requires the +stat privilege). Another client may use find with a filter that inspects a file's name (which requires +path, but not +stat). For both clients, the contract guarantees that the implementation of find itself cannot use either the +stat or +path privileges, even though it invokes the functions filter and cmd.

2.5 Interaction with ambient authority

Figures 3, 4, and 5 show SHILL scripts that consume and use capabilities. But where do capabilities come from? SHILL is intended for use with commodity operating systems, and so we must provide a mechanism to transition from the ambient world of the operating system to SHILL's capability-safe world (Goal 5).

To that end, in addition to the capability-safe scripts we have described so far, users of SHILL scripts write *ambient scripts* which inherit the authority of the invoking user and are *not* capability safe. Ambient scripts are used to create capabilities and pass them to functions that capability-safe scripts provide. Consequently, the language of ambient scripts is extremely restricted: ambient scripts contain straight line code that can import capability-safe scripts, create capabilities for resources using file paths and other global names, and call

```
1  #lang shill/ambient
2
3  require shill/native;
4  require "jpeginfo.cap";
5
6  root = open-dir("/");
7  wallet = create_wallet();
8  populate_native_wallet(wallet,root,
9     "~/Downloads/jpeginfo",
10    "/lib:/usr/local/lib",
11    pipe_factory);
12
13 dog = open-file("~/Documents/dog.jpg");
14 jpeginfo(wallet,stdout,dog);
```

Figure 6: Ambient script to call jpeginfo

functions exported by capability-safe scripts. Ambient scripts are brief and delegate all interesting tasks to the capability-safe scripts they import. Also, capability-safe scripts cannot import ambient scripts, which ensures that capability-safe scripts cannot use ambient scripts to obtain additional capabilities. Ambient scripts must reason carefully about their interaction with untrusted scripts. Contracts and capabilities help with this.

Figure 6 shows an ambient script that creates appropriate capabilities and then invokes the jpeginfo function from the script in Figure 4. The annotation #lang shill/ambient on line 1 indicates that this is an ambient script.[4] Line 3 loads a SHILL library script that helps create capability wallets. Line 4 loads the capability-safe script from Figure 4.

Lines 8–11 create an appropriate capability wallet to run `jpeginfo` by calling the trusted standard library function populate_native_wallet. Line 13 creates a capability for ~/Documents/dog.jpg. The capability has all privileges that the invoking user is allowed for this file; when the capability passes through a capability contract, it loses all privileges except those stated in the contract. Line 14 invokes jpeginfo with the capability wallet, a capability to standard out, and the capability to `dog.jpg`.

3 Implementation

We have implemented a prototype of SHILL as a kernel module and set of userspace tools for FreeBSD 9.2. The userspace tools include the SHILL compiler, runtime, and standard library. The kernel module implements capability-based sandboxes and provides capability-safe versions of several POSIX system calls.

[4] Capability-safe scripts have the annotation #lang shill/cap on the first line; we omitted this annotation in Figures 3, 4, and 5.

3.1 Language

We implement the SHILL language as an extension to Racket [9] using Racket's macro system and tools for building languages [39]. Prototyping SHILL in this way allows us to use Racket functionality where it meets our security requirements. In particular, we used Racket's contract mechanism to implement SHILL contracts.

A distinguishing feature of SHILL is capability safety: access to resources occurs only through capabilities, and creation of capabilities is limited. To achieve capability safety at the language level, we (1) provide language-level capabilities and capability contracts; (2) restrict the expressiveness of the language; and (3) provide a capability-based language runtime for SHILL.

3.1.1 Capabilities and their Contracts

Capabilities in the SHILL language are object-like values that encapsulate low-level capabilities such as file descriptors or sockets. Each operation on a capability is implemented by calling the corresponding operation on the low-level capability. Different kinds of capabilities support different operations. For example, supported operations on files and pipes include reading, writing, and changing modes. Directories also have capabilities for listing, adding, or removing directory entries. Each operation has a corresponding privilege that can be present or absent on a given capability. In total, SHILL has twenty-four different privileges for filesystem capabilities and seven different privileges for sockets. Socket privileges are further refined by connection type.

We chose privileges and operations to align closely with the operations that our capability-based sandbox can interpose on, so that we can ensure that giving a capability to a sandbox conveys the same authority as giving that capability to a SHILL script. There are two kinds of SHILL capabilities that do not encapsulate a system resource directly: the pipe_factory and socket_factory capabilities. These capabilities encapsulate, respectively, the right to create new pipes or sockets. The pipe_factory capability has a create operation that returns a pair of pipe ends. Each pipe end is a file capability. In our prototype implementation, SHILL scripts cannot create or manipulate sockets directly (which can be addressed by adding built-in functions for socket operations to the language). We do restrict a sandbox's permitted socket operations: a sandbox must possess a socket_factory capability to be allowed to create and use sockets.

We implement SHILL contracts using Racket contract combinators [8, 7] that create proxies [38] for capabilities, allowing us to interpose on operations and check privileges before allowing an operation. These proxies also store information about the privilege restrictions each contract imposes.

Resource	Language	Sandbox
Directories, files, links	Capabilities	Capabilities
Pipes	Capabilities	Capabilities
Character Devices	Capabilities	Capabilities†
Sockets (IP,Unix)	Capabilities	Capabilities
Sockets (other)	Denied	Denied
Processes	ulimit‡	Confinement
Sysctl	Denied	Read-only
Kernel environment	Denied	Denied
Kernel modules	Denied	Denied
POSIX IPC	Denied	Denied
System V IPC	Denied	Denied

Figure 7: System resources and how each is protected in the SHILL language and capability-based sandboxes.
†: In our prototype, character devices are only partially controlled by capabilities, see Section 3.2.3.
‡: SHILL allows calls to the exec function to specify ulimit parameters for the child process.

3.1.2 Restricting the SHILL language

To achieve capability safety in SHILL, we carefully choose which language features and libraries of Racket are available in SHILL. We allow access to certain Racket libraries, such as the regular expression library, but prevent access to all others, including Racket's system library and Racket's macro system. SHILL scripts are allowed to import only SHILL capability-safe scripts.

The ambient SHILL language (see Section 2.5) has further restrictions: it may not do anything other than import capability-safe SHILL scripts, create strings and other base values, define (immutable) variables, and invoke functions. However, unlike the capability-safe SHILL language, it may create capabilities using ambient authority.

3.1.3 Capability-based runtime

We implemented a capability-based language runtime for SHILL that provides operations to access files and other resources through file descriptors. (The Racket libraries for accessing files and other resources rely on ambient authority, and are thus not suitable for our use.) File descriptors provide unforgeable tokens that can serve as low-level capabilities for directories, files, links, pipes, sockets, and devices. Our capability-based runtime provides wrappers for the `*at` family of system calls which provide a file-descriptor based interface to common operations like opening, reading, and writing files. Our runtime further restricts these system calls by requiring that arguments that specify sub-paths contain only a single component. For example, the pathname argument to `openat` may be `alice` but not `alice/dog.jpg` or `../bob`. Our runtime also provides wrappers for standard system calls which can be used by SHILL's ambient language to create capabilities for system resources.

Most but not all FreeBSD system calls that manipulate the filesystem have a version that consumes file descriptors rather than paths. The `linkat`, `unlinkat`, and `renameat` system calls use file descriptors to designate target directories, but rely on paths to designate files. Thus, a call to `linkat` can not be guaranteed to link to the correct file without risking a time-of-check-to-time-of-use vulnerability. Our kernel module adds three system calls to address these deficiencies: `flinkat`, which installs a link to a file in a directory given file descriptors for both the file and the directory; `funlinkat`, which takes a name and file descriptors for a file and a directory and removes the link at the given name if it refers to the file; and `frenameat`, which is similar to `funlinkat` but also installs a link to the file in a target directory. The module also provides a version of `mkdirat` that returns a file descriptor for the newly created directory.

We also add a new `path` system call that attempts to retrieve an accessible path for a file descriptor from the filesystem's lookup cache. SHILL uses this system call to provide a relatively robust mechanism to translate SHILL capabilities into paths to provide as arguments to sandboxed executables. If the `path` system call fails, SHILL uses the last known path at which the file was accessible.

Our prototype implementation of SHILL does not provide support for all system resources. Interaction with resources that do not correspond to capabilities is either restricted or denied entirely. Figure 7 lists system resources and how SHILL controls access to these resources in the language and in capability-based sandboxes. There is no fundamental obstacle to providing capability support for all resources, though doing so would require additional modifications to the system call interface. For example, we would need to provide a low-level capability for processes, similar to Capsicum's *process descriptors* [43].

3.1.4 Standard Library

SHILL's standard library provides a number of capability-safe scripts that help programmers write SHILL scripts. The filesys script provides capability-based functions that emulate common tasks such as resolving paths and symlinks. The io script provides printf-like wrappers around write and append for formatted output. The contracts script provides abbreviated definitions of common contracts. For example, a programmer can specify the contract readonly rather than the more verbose

```
dir(+read-symlink,+contents,+lookup,
    +stat,+read,+path) ∨ file(+stat,+read,+path).
```

Capability wallets Recall that capability wallets are maps from strings to lists of capabilities that help automate and simplify the discovery, packaging, and use of capabilities to invoke executables in sandboxes. SHILL provides functions for creating and using capability wallets. For example, the native script in the standard library provides two functions for using native wallets to invoke executables (as in Figures 4 and 6): populate_native_wallet and pkg_native.

Function populate_native_wallet helps create a native wallet. Its arguments include path specifications for where to search for executables and libraries (i.e., colon-separated strings, analogous to environment variables `$PATH` and `$LD_LIBRARY_PATH`), and a directory capability to use as a root for the path specifications. In addition, it takes a map (of strings to lists of strings) from known libraries to the file resources those libraries depend on. Function populate_native_wallet uses the directory capability to resolve the path specifications (i.e., converts the lists of strings to lists of capabilities), and places these capabilities in a native wallet. It also resolves the known dependencies (i.e., the map from known libraries to the file resource path names) into a map from strings to lists of capabilities, and places the resolved map into the native wallet.

Function pkg_native takes a native wallet and a file name (of an executable file) and searches the path capabilities in the native wallet for a capability for the executable. The function then invokes `ldd` to obtain a list of libraries that the executable depends on, and searches the library-path capabilities for capabilities for the required libraries. Once these capabilities are gathered, pkg_native uses the map of known dependencies to gather additional capabilities needed to run the executable. Function pkg_native then returns a function that encapsulates a call to exec with all capabilities needed to run the executable. Figure 4 shows an example script that uses pkg_native.

3.2 Capability-based sandbox

The SHILL sandbox is implemented as a policy module for the TrustedBSD MAC Framework [41] (hereafter, "the MAC framework"). The MAC framework allows FreeBSD's access control mechanisms to be extended with third-party mandatory access control policies by mediating access to sensitive kernel objects and invoking access control checks specified by third-party policy modules. The framework also provides a policy-agnostic mechanism for attaching security labels to kernel objects. Mechanisms with similar functionality are available on Linux and Apple's OS X.

3.2.1 Session lifecycle

Each process executing in a SHILL sandbox is associated with a *session*. Processes in the same session share the same set of capabilities and can communicate via sig-

nals. Processes spawned by a process in a session are by default placed in the same session. However, sessions are hierarchical: a sandboxed process inside session S_1 can spawn a process inside a new session S_2, which has fewer capabilities that S_1. This allows SHILL-aware executables to further attenuate their privileges.

New sessions are created by invoking the system call `shill_init`, which creates a session and associates it with the current process. A new session initially has no capabilities of its own. Capabilities possessed by the parent session can be granted to the new session until the process invokes the `shill_enter` system call. Once `shill_enter` is called, the session allows only operations permitted by capabilities it was granted explicitly.

3.2.2 From capabilities to MAC labels

Each system resource protected by a SHILL capability corresponds to an underlying kernel object: a filesystem vnode, pipe, device, or socket. Using the MAC framework's ability to attach labels to kernel objects, SHILL labels these kernel objects with a *privilege map*: a map from sessions to sets of privileges. A privilege map records the privileges that each session has for the given kernel object. Privileges in the privilege map correspond directly to privileges of SHILL capabilities.

When a SHILL script calls exec, the SHILL runtime sets up a sandbox by forking a new process, creating a new session, and granting the session the capabilities passed to exec. It then calls `shill_enter` before transferring control to the executable.

When a sandboxed process invokes a system call relevant to a resource protected by SHILL, we use the privilege map for that resource to check whether the process's session has sufficient privileges for the operation. If there are insufficient privileges, the system call aborts with an error but the process is otherwise allowed to continue.

Derived capabilities In the SHILL language, some operations on SHILL capabilities yield derived capabilities. For example, using a directory capability, a script might obtain capabilities for children of the directory, or might obtain a capability for a new file created in that directory. In the sandbox, we track these derived capabilities by updating privilege maps in response to operations on kernel objects. To enable this, we extended the MAC framework with two additional hooks: `mac_vnode_post_lookup` and `mac_vnode_post_create`. These entry points are invoked after a lookup or create operation completes successfully, and allow the SHILL policy module to update the privilege map on the resulting vnode. For example, if session S has privilege +lookup with {+stat,+path} on a vnode for a directory `d`, and a process in that session successfully invokes system call `openat(d, "child", flags)`, then the SHILL policy module updates the privilege map for the vnode for file `child` to add privileges +stat and +path for session S.

Path traversal To achieve fine-grained confinement in the filesystem, SHILL scripts are not permitted to follow the "`..`" entry of a file or directory capability. However, simply disallowing use of "`..`" in SHILL's capability-based sandboxes would break many existing programs. Instead, the sandbox allows any lookup operation on a directory if the session has the +lookup privilege, but only propagates privileges when the lookup would have been permitted in the SHILL language, that is, when the directory entry requested is not "`..`".[5]

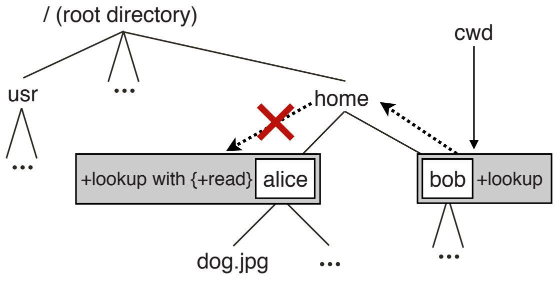
Example Consider a sandboxed process attempting to call `open("../alice/dog.jpg", O_RDONLY)` from the current working directory `/home/bob`. This system call invokes a series of low-level `lookup` operations on filesystem objects to resolve the path and create a file descriptor for the designated resource.

Figure 8 depicts the process of completing these operations in a SHILL sandbox. Shaded boxes around nodes in the file system denote privileges held by the current session. The current working directory is indicated with a solid arrow. Dashed arrows represent low-level lookup operations, and a dashed box around a node represents privileges propagated in response to a lookup operation.

In the left diagram, the current session has a capability to the vnode corresponding to `/home/alice` and a capability to the current working directory. The first operation (lookup "`..`" in `/home/bob`) is permitted because the process has the +lookup privilege, but privileges are not propagated to the vnode for `/home`. Thus, the second operation (lookup `alice` in `/home`) fails because the session does not have the necessary privileges. The `open` system call returns `EACCES` to indicate that the process had insufficient privileges.

The right diagram considers the same scenario, but where the session also has a +lookup privilege to the directory `/home`. In this case, the session is permitted to look up `alice` in `/home`. The final operation (lookup `dog.jpg` in `/home/alice`) also succeeds. These two lookups propagate privileges from the parent nodes to the results of the lookup. Looking up `dog.jpg` in `/home/alice` grants the session the privilege +read on the vnode representing `dog.jpg`, since the session had privilege +lookup with {+read} on the vnode for `/home/alice`. Thus, the call `open("../alice/dog.jpg", O_RDONLY)` succeeds.

[5] We also do not propagate privileges when the directory entry is "`.`", since this can lead to privilege amplification. For example, if session S has only the privilege +lookup with +stat on directory `d`, then calling `openat(d, ".", flags)` would give S the +stat privilege on `d`.



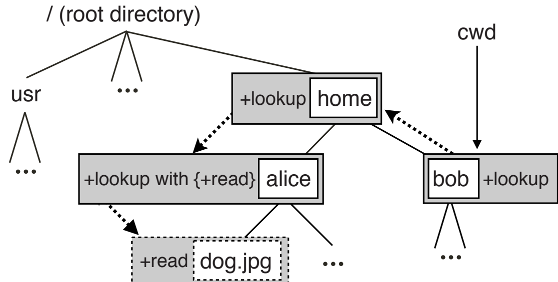


Figure 8: Resolving system call `open("../alice/dog.jpg", O_RDONLY)` in a capability-based sandbox. Left: the session has privileges for `/home/alice` and `/home/bob`, but not `/home`, so the operation fails. Right: the session also has a lookup privilege for `/home`, so the operation succeeds and the lookup privilege on `/home/alice` is propagated to `/home/alice/dog.jpg`.

Note that unlike SHILL scripts, sandboxed executables are vulnerable to confused deputy attacks if they allow clients to specify resources with paths rather than, e.g., file descriptors. However, the authority of the sandboxed execution is still limited by the capabilities it is granted.

Avoiding privilege amplification In the SHILL language, capabilities both designate resources and confer privileges. As a consequence, it is possible to have two separate capabilities to the same resource with different privileges. These separate capabilities may confer less privilege than a single capability with the combined privileges. For example, consider a pair of capabilities to create a network socket, one with sufficient privileges to send but not receive messages at a particular port, and one with sufficient privileges to receive but not send messages on the same port. Because only a single socket can be bound to a port, a program with these capabilities must choose to either send or receive messages.

Since in the SHILL language, scripts cannot combine capabilities, possessing multiple capabilities for the same resource does not lead to privilege amplification. In the capability-based sandbox, however, to avoid privilege amplification the sandbox must prevent two separate capabilities to the same object from being combined to allow additional operations.

For file system operations that create new objects (e.g., creating new files or directories), SHILL requires that a session is never granted conflicting privileges to the same object. For example, if session *S* currently has privilege +create-file with {+read,+stat,+path} for a directory *d*, (i.e., the privilege to create read-only files), and due to a lookup from the parent directory we want to propagate privilege +create-file with {+write}, we would *not* merge these privileges, i.e., we would *not* give *S* the privilege +create-file with {+write,+read,+stat,+path}. While more sophisticated techniques to track privileges are possible, we have found that this conservative approach to prevent privilege amplification works well in practice, and does not break functionality of any of our case studies.

Process interaction The SHILL language provides limited support for operations on processes: SHILL does not have capabilities to control the creation of processes, process synchronization, interprocess communication, etc.

Within capability-based sandboxes, we enforce a simple security policy for operations related to processes: processes in a session can only interact with processes in the same session or a descendent session. A process in a sandbox cannot debug, send signals to, or wait for a process outside of its session.

Debugging SHILL provides several tools for debugging processes running in SHILL sandboxes. First, there is a command-line tool for running a single shell command with capabilities specified in a policy file. Second, for all SHILL sandboxes, logging can be enabled and viewed by privileged users. The log records all of the capabilities and privileges granted during a session in addition to all operations that were denied because of insufficient privileges. Using the command-line tool, a session can be created in debugging mode, which automatically grants the necessary privileges if an operation would fail. We found that running programs in a debugging sandbox and then viewing the logs was a useful starting point for identifying necessary capabilities to provide to a SHILL script. However, as we developed additional standard library support to run common executables, this became less necessary. In most cases, the utilities in the standard library automate the retrieval and collection of capabilities needed to run an executable.

3.2.3 Limitations

SHILL's capability-based sandboxes rely on the MAC framework to implement access control checks based on

capabilities. Thus, the granularity of the MAC framework's mechanism determines the granularity at which our sandboxes protect resources. For example, the MAC framework exposes a single entry point for operations that write to filesystem objects, so we cannot distinguish write and append operations. Conservatively, we enforce that to write (or append) to a file, a session must have both the +write and +append privileges for the file. (Note that in SHILL scripts, privileges can be enforced at fine granularity, since capability safety in scripts relies on language abstractions, not on the MAC framework.)

The MAC framework does not interpose on `read` or `write` operations on character devices. Thus, while the SHILL language exposes `stdin`, `stdout`, and `stderr` as file capabilities and enforces restrictions on how they can be used, sandboxed processes can bypass these restrictions if one of these capabilities abstracts a pseudo-terminal or other device. This limitation is not fundamental and can be resolved by adding entry points to the MAC framework around unprotected operations. It can be mitigated by not granting capabilities to such devices to sandboxes.

4 Evaluation

We evaluate the expressiveness of SHILL through four case studies: a grading script for a programming assignment, a package management script for the GNU Emacs editor, sandboxing the Apache web server, and a find and execute task similar to the example in Section 2. We measure the performance of SHILL via case studies and microbenchmarks. Our evaluation indicates that (1) SHILL is a practical security tool for typical system tasks, (2) SHILL can provide fine-grained security guarantees when scripts are used to compose untrusted software and, (3) its performance cost is pay-as-you-go, i.e., weak security guarantees incur little overhead.

4.1 Case studies

Grading submissions We used SHILL to securely grade student submissions written in OCaml for an undergraduate programming languages course. As a baseline, we wrote a 61-line Bash script that compiles the OCaml source code of each submission and runs the compiled program against a test suite. Results of executing student submissions against the test suite are recorded in a grading directory, one file per student.

With minimal effort, we secured this Bash script in a SHILL sandbox. The capability-safe script that executes the Bash script in a sandbox is 22 lines, of which 14 are the contract for the script. The ambient script that invokes capability-safe script is also 22 lines. The contract guarantees that the grading script can at most: read files in directories containing student submissions and tests; create, modify, and delete new files in a working directory and the output directory; and access the system resources needed to run the compiler and compiled programs.

To demonstrate the finer-grained guarantees of SHILL, we also wrote a version of the grading script exclusively in SHILL. The capability-safe grading script is 78 lines of code, of which six are the script's contract. The ambient script that invokes it is 16 lines. The SHILL script provides all the security guarantees of the sandboxed Bash script, and also ensures that while grading a student's submission, no other student's submission, working-directory files, or results file can be accessed.

The capability-safe SHILL script was developed by manually translating and modifying the original Bash script. String-based references to files were replaced with appropriate capabilities. Calls to programs like `gmake`, `diff`, and `ocamlrun` were replaced with calls to the SHILL standard library to package and execute those programs. To enable this, the ambient script creates a native_wallet initialized with a standard `PATH` and `LD_LIBRARY_PATH`. Contracts for the capability-safe SHILL script ensure that each student's grading file is isolated from other students and that students' programs can't directly modify their grade file. These fine-grained guarantees—which the Bash script does not provide—are achieved by ensuring that the contract on the grading directory allows only the creation of new append-only files, and the functions that compile and execute a student's submission are given no capabilities to other students' grading files.

In developing this script, we debugged several cases where the script had too few privileges to run successfully. In one case, we wrote too restrictive a contract for the submissions directory, forgetting the +lookup privilege. The resulting contract failure indicated which argument had insufficient privileges. After verifying that this privilege was necessary and did not compromise the security guarantees, we fixed the script. We encountered two issues with sandboxed executables. First, the wallet used to launch executables was missing some necessary capabilities: when trying to compile students' submissions, `ocamlc` reported that it was unable to read a file in `/usr/local/lib/ocaml`. Investigating, we realized that OCaml searches for libraries in this directory. Adding the directory to the wallet as a dependency for OCaml executables fixed the issue but revealed another: `ocamlyacc` could not write to `/tmp`. After adding a capability to `/tmp` when invoking `gmake`, the script ran successfully. To ensure isolation between different invocations of `gmake`, we used a contract on the `/tmp` capability to specify that sandboxed processes can only read, modify, or delete files or directories they create.

Package Management We used SHILL to write an installation script for GNU Emacs (similar to what may be found in a package manager). The script provides functions to download, compile, install, and uninstall Emacs. Unlike a typical package manager, the script has a detailed security interface for each function. For example, only the function for downloading the source code can access the network, and only the install function can write to the intended installation directory. In addition, the install function is restricted from reading, altering, or removing any existing files in the installation directory, and the uninstall function's contract gives a list of files that it is permitted to remove. The package manager comprises 114 lines of ambient code, and 91 lines of capability-safe code, of which 45 specify contracts.

Apache web server To showcase how SHILL handles networking applications, we used SHILL to develop a sandbox for the Apache webserver, version 2.2. We tested the performance of the web server by using the Apache Benchmark tool to download a 50MB file served by Apache five thousand times using up to 100 concurrent connections. In addition to its required libraries, the script's contract gives the webserver read-only access to configuration files and web content directories, the ability to create and use sockets, and write-only access to log files. The ambient script is 27 lines, and the capability-safe script is 30 lines, of which 20 lines are contracts.

Find As another example of how programmers can use SHILL to gradually strengthen the guarantees of scripts, we developed two versions of a SHILL script for a find and execute task. Our scripts find all files with extension .c in the BSD source tree that contain the string "`mac_`", the prefix on entry points for the MAC framework. Completing this task requires visiting 57,817 files and invoking `grep` on the 15,376 files with extension .c.

The simpler version is a SHILL script that launches a sandbox for the command

```
find /usr/src -name "*.c" \
            -exec grep -H mac_ {} \;
```

The ambient script is 11 lines and calls a 27-line capability-safe script, of which 5 lines are contracts. The contract ensures that the sandbox has access only to `/usr/src` and files necessary to run `find` and `grep`.

The second version uses the find function (Figure 5) to find files with the extension .c and invokes `grep` in a sandbox for each matching file. In addition to the guarantees of the previous version, this script provides the fine-grained guarantee that the files that `grep` operates on are exactly the files selected by the find function. Note that our first script does not provide this guarantee: paths passed to `grep` may resolve to different files. The ambient script is 9 lines, and the capability-safe script is 60 lines, of which 11 are contracts.

4.2 Performance Analysis

Our prototype implementation focuses on providing fine-grained security guarantees, and we have not yet optimized performance. However, to verify that the performance costs of SHILL are commensurate with the security guarantees, we use the case studies as benchmarks. We also develop benchmarks for sub-tasks of the Emacs installation script (download, untar, configure, make, make install, make uninstall). For each benchmark, we derive a command line invocation to achieve the same task as the case study outside of SHILL (if such a command was not already part of the case study).

We measured the performance of each benchmark in three different configurations. The "Baseline" configuration executes the command on FreeBSD without the SHILL kernel module installed. The "SHILL installed" configuration executes the command with the kernel module installed (but not active). The "Sandboxed" configuration uses a SHILL script to create a sandbox for the command. Where applicable, we also executed a "SHILL version" of the case study that replaces the command.

We ran each configuration of each benchmark 50 times and computed the mean time to completion along with a 95% confidence interval. The performance measurements were conducted on a six core, 3.33GHz Xeon server with 6GB of RAM running FreeBSD 9.2. Figure 9 presents the results. We compare performance with "Baseline" using a two-sided t-test on the difference in mean run time. Statistical significance was determined at the 0.05 level after a Bonferroni correction for multiple hypothesis testing within each benchmark.

First, observe that the overhead of our system for programs that are not secured by SHILL scripts is negligible. Second, the slowdown for "Sandboxed" and "SHILL version" configurations ranges from negligible to 1.21×, except for a few extreme cases: the "Sandboxed" configurations of the Download and Uninstall benchmarks and the "SHILL version" of the Find benchmark. These tasks are 1.73×, 6.61×, and 6.01× slower than the baseline, respectively. We explore these high overheads below. Third, the SHILL version of the package management benchmark has no significant overhead and the SHILL version of the grading script is only 1.13× slower, despite the finer-grained guarantees these scripts provide.

Profiling To better understand the performance of SHILL, we profiled the "SHILL version" configurations of the Grading and Find benchmarks, and the "Sandboxed" configurations of Download and Uninstall. We inserted instrumentation to measure the total execution

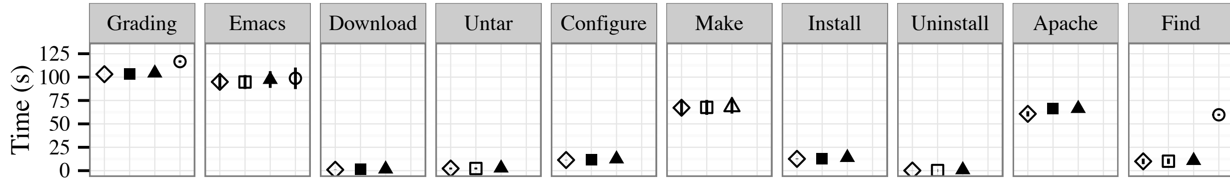


Figure 9: Performance of SHILL for a variety of tasks. Running time is given for the "Baseline" (◇), "SHILL installed" (□), "Sandboxed" (△), and "SHILL version" (◯) configurations. 95% confidence intervals are indicated by vertical bars. Bars may be hidden by plotting symbols when confidence intervals are small. Configurations that differ significantly from "Baseline" are filled (e.g., ■).

time, Racket startup (which includes script compilation, and starting the runtime), setup of sandboxes, and sandboxed execution for each benchmark. Figure 10 shows the results. Remaining time (i.e., time not spent on Racket startup, sandbox setup, or sandboxed execution) is time spent executing SHILL scripts, including contract checking. We used a Racket profiler [36] to estimate how SHILL's features affect the running time. Most time spent executing SHILL scripts is in capability-safe scripts (more than 99% for both Find and Grading) and in particular checking contracts (86% for Find and 87% for Grading). The contract on the result of pkg-native accounts for almost all contract checking time (92% and 93% of contract checking time for Find and Grading respectively) because it is checked once per sandbox. (The remaining time for the Download and Uninstall benchmarks was insufficient for the profiler to produce meaningful data.)

For these benchmarks, most time outside of sandboxed execution is spent enforcing security guarantees: checking contracts and setting up sandboxes. The Grading benchmark creates 5,371 sandboxes, Find creates 15,292, Uninstall creates one, and Download creates two (one for pkg-native and one for the executable, `curl`). Grading and Find create many sandboxes, each of which takes a relatively small amount of time to set up and a relatively small amount of time to check the contract from pkg-native. Racket startup cost is responsible for the high overhead of Download and Uninstall. The high overhead of Find is due to contract checking and sandbox setup, but also due to high sandboxed execution time. A small portion of the latter cost is due to overhead on system call interposition for privilege checking (see microbenchmarks below). We conjecture that the remaining cost stems from the high number of short-lived sandboxes that Find creates, which causes contention between threads using privilege maps and the kernel's asynchronous cleanup of expired SHILL sandbox sessions.

Microbenchmarks To understand the overhead added to system calls due to privilege checking during sandboxed execution (see Section 3.2.2), we evaluated microbenchmarks for several representative system calls under both the "SHILL installed" and "Sandboxed" configurations. The `pread`-1B microbenchmark reads one byte from an opened file; `pread`-1MB reads 1 megabyte. The `create-unlink` microbenchmark creates a new file, closes, and unlinks it. The `open-read-close` benchmarks open a file, reads one byte, and closes it. In one version of this benchmark, the path argument to `open` has length one, and in the other it has length five (i.e., the file is nested in 4 subdirectories).

	Uninstall	Download	Grading	Find
Total time	0.82 s	1.66 s	116.38 s	61.20 s
Racket startup	0.65 s	0.63 s	0.92 s	0.65 s
Sandbox				
setup	0.01 s	0.01 s	6.98 s	18.04 s
execution	0.14 s	0.96 s	104.09 s	27.61 s
Remaining time	0.03 s	0.07 s	4.39 s	14.90 s

Figure 10: Performance breakdown of four benchmarks.

Operation	SHILL Installed	Sandboxed	Difference
`pread`-1B	516 ± 80 ns	560 ± 64 ns	44 ± 102 ns
`pread`-1MB	199 ± 4 ms	202 ± 6 ms	3 ± 7 ms
`create-unlink`	13 ± 3 ms	14 ± 4 ms	1 ± 4 ms
`open-read-close`			
1 lookup	3.7 ± 0.4 ms	4.0 ± 0.4 ms	0.3 ± .6 ms
5 lookups	5.3 ± 0.3 ms	6.4 ± 0.5 ms	1.1 ± 0.6 ms

Figure 11: Overhead of SHILL for microbenchmarks.

We timed one million iterations of each microbenchmark, except for `pread`-1MB, which was executed one thousand times. Figure 11 shows the mean execution time and 95% confidence intervals. All differences were statistically significant. The overhead of executing system calls in a SHILL sandbox ranges between 18% (`open-read-close`, 5 lookups) and 1% (`pread`-1MB). For the `open-read-close` benchmarks, further experiments (not shown) indicate that overhead increases linearly in the length of the path (i.e., linearly with the number of lookup system calls required).

5 Related work

Much research is devoted to controlling the authority of untrusted software and applying the Principle of Least Privilege (POLP), spanning operating system design, systems security, and programming languages.

Operating Systems Capabilities are a well-known and effective mechanism to support POLP. Capability-based operating systems [6] such as KeyKOS [11, 5], EROS [34], Coyotos [33] and PSOS [29] use operating system and hardware capabilities to limit the authority of users and processes. Numerous microkernels inspired by the L4 family [19] employ capabilities as an access control mechanism [4, 13, 18]. HiStar [44] and Asbestos [44] track information flow to enforce fine-grained security policies. SHILL is not an operating system and is built on a commodity operating system. However, it shares similar goals and draws inspiration from these novel systems. For instance, the source of certain kinds of capabilities in KeyKOS is the *command system*: the only program in the system with ambient access to a user's directory. SHILL's ambient scripts serve the same purpose.

Capsicum [43] extends the FreeBSD operating system with capabilities but requires programs to be rewritten to use the capability-based interfaces in order to make use of capability mode. By contrast, SHILL's capability-based sandbox does not require executables to be aware of capabilities. In addition, SHILL capabilities are more expressive than Capsicum capabilities; for example, a SHILL capability can express the permission to create files in a directory and delete only files that were created with the capability.

Systems security Laminar [30] integrates operating system and programming language abstractions to enforce decentralized information flow control (DIFC). Its high-level architecture resembles that of SHILL. However, Laminar provides fine-grained security only for programs that use Laminar's security abstractions, and does not provide declarative security specifications. Hails [10] uses declarative information-flow control policies as a mechanism for composing mutually distrusting web applications. Unlike SHILL, it provides limited support for securing legacy applications. Flume [17] uses a user-space reference monitor for DIFC at the granularity of operating system abstractions. While both SHILL and Flume can enforce security restrictions on untrusted applications, SHILL uses capabilities and contracts rather than DIFC labels.

A plethora of sandboxing tools have been developed for commodity operating systems, including SELinux [20], Seatbelt [42], AppArmor [1], GrSecurity [35], LXC [3], and Docker [2]. Unlike SHILL, these sandboxes deny or grant access based on a profile rather than a programmable capability-based interface. Mbox [15] and TxBOX [14] create sandboxes with transactional semantics that can reverse the effects of misbehaving processes, but enforce strong isolation between sandboxed processes and the rest of the system. Notably, programs running in a SHILL sandbox are not isolated from the rest of the system. For example, in our Apache case study, concurrently executing programs can dynamically add new web content or view logs as they are generated. Many of these sandboxes require root privileges, but some are available to all users [15]. PLASH [32] is a capability-based interactive shell for creating sandboxes in which to execute shell commands, similar to SHILL's `exec`. All of these tools lack the reasoning principles SHILL provides for composing multiple sandboxes together.

Programming languages The use of language-level capabilities to support POLP has a long history [28]. The E programming language [26] is a seminal *object capability language*, where capabilities are object references. CapDesk [40, 37] is a desktop shell for launching applications written in E. Applications are granted limited authority initially and can gain more capabilities through *powerboxes*, which mediate requests for authority from the application to the user. In contrast to SHILL, CapDesk does not have a scripting interface and applications launched by CapDesk must be capability-aware and designed to work with the CapDesk framework.

Joe-E [22] restricts Java to an object-capability-safe subset. Similarly, Caja [27] introduces an object-capability-safe subset of JavaScript. Maffeis et al. [21] prove that these subsets are indeed capability safe. Unlike other capability-safe languages, SHILL targets a particular domain (shell scripting) instead of general programming and that it uses contracts to manage capabilities instead of capability-based design patterns [26].

Acknowledgments

We thank Dan Bradley for his contributions to an early version of this work, and Jennifer Kirk for her help with statistical analysis. We are grateful to Leif Andersen, Vincent St-Amour, and Matthias Felleisen for their help profiling SHILL code. We thank Eddie Kohler, the Programming Languages Group at Harvard, and the reviewers for their helpful comments. Many thanks to Frans Kaashoek for his thoughtful shepherding. This research is supported by the Air Force Research Laboratory.

References

[1] Apparmor. https://wiki.ubuntu.com/AppArmor.

[2] Docker. https://www.docker.io.

[3] LXC. https://linuxcontainers.org.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 29–44, 2009.

[5] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*. USENIX Association, 1992.

[6] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9 (3):143–155, 1966.

[7] R. B. Findler and M. Blume. Contracts as pairs of projections. In *Proceedings of the 8th International Symposium on Functional and Logic Programming*, pages 226–241, 2006.

[8] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Programming*, pages 48–59, 2002.

[9] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang.org/tr1/.

[10] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60. USENIX, 2012.

[11] N. Hardy. KeyKOS architecture. *Operating Systems Review*, 19 (4):8–25, 1985.

[12] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[13] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[14] S. Jana, D. E. Porter, and V. Shmatikov. TxBox: Building Secure, Efficient Sandboxes with System Transactions. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, May 2011.

[15] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, pages 139–144, Berkeley, CA, USA, 2013. USENIX Association.

[16] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Conference on Hot Topics in Operating Systems*, page 21, Berkeley, CA, USA, 2005. USENIX Association.

[17] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334, 2007.

[18] A. Lackorzynski and A. Warg. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, 2009.

[19] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 237–250, 1995.

[20] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[21] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symposium on Security and Privacy*, pages 125–140, May 2010.

[22] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2010.

[23] B. Meyer. Design by contract. In *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

[24] B. Meyer. Applying “Design by Contract”. *Computer*, 25(10): 40–51, 1992.

[25] M. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University, 2003.

[26] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[27] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. Google white paper. http://google-caja.googlecode.com, 2008.

[28] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, January 1973.

[29] P. G. Neumann and R. J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 208–216, Dec 2003.

[30] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 63–74, 2009.

[31] J. H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974. ISSN 0001-0782.

[32] M. Seaborn. PLASH: the principle of least authority shell, 2007. http://www.cs.jhu.edu/~seaborn/plash/html/.

[33] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In *Proceedings of the NICTA Invitational Workshop on Operating System Verification*, pages 1–19. USENIX, 2004.

[34] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, 1999.

[35] B. Spengler. Grsecurity ACL docmentation v1.5, 2003. http://grsecurity.net/gracldoc.htm.

[36] V. St-Amour and M. Felleisen. Feature-specific profiling. Technical Report NU-CCIS-8-28-14-1, Northeastern University, August 2014.

[37] M. Stiegler and M. Miller. A capability based client: The DarpaBrowser. Technical Report BAA-00-06-SNK, COMBEX Inc., June 2002.

[38] T. S. Strickland, S. Tobin-Hochstadt, R. Findler, and M. Flatt. Chaperones and impersonators. In *Proceedings of the ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 943–962, 2012.

[39] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 132–141, 2011.

[40] D. Wagner and D. Tribble. A security analysis of the Combex DarpaBrowser architecture. Online at: http://www.combex.com/papers/darpa-review/, Mar. 2002.

[41] R. Watson and C. Vance. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *In USENIX Annual Technical Conference*, pages 285–296, 2003.

[42] R. N. M. Watson. A decade of OS access-control extensibility. *Communications of the ACM*, 56(2):52–63, 2013.

[43] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security Symposium*, pages 29–46. USENIX Association, 2010.

[44] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, 2006.

GPUnet: Networking Abstractions for GPU Programs

Sangman Kim Seonggu Huh Yige Hu
Xinya Zhang Emmett Witchel
The University of Texas at Austin

Amir Wated Mark Silberstein[1]
Technion – Israel Institute of Technology

Abstract

Despite the popularity of GPUs in high-performance and scientific computing, and despite increasingly general-purpose hardware capabilities, the use of GPUs in network servers or distributed systems poses significant challenges.

GPUnet is a native GPU networking layer that provides a socket abstraction and high-level networking APIs for GPU programs. We use GPUnet to streamline the development of high-performance, distributed applications like in-GPU-memory MapReduce and a new class of low-latency, high-throughput GPU-native network services such as a face verification server.

1. Introduction

GPUs have become the platform of choice for many types of parallel general-purpose applications from machine learning to molecular dynamics simulations [3]. However, harnessing impressive GPU computing capabilities in complex software systems like network servers remains challenging: GPUs lack software abstractions to direct the flow of data within a system, leaving the developer with only low-level control over I/O. Therefore, certain classes of applications that could benefit from GPU's computational density require unacceptable development costs to realize their full performance potential.

While GPU hardware architecture has matured to support general-purpose parallel workloads, the GPU software stack has hardly evolved beyond bare-metal interfaces (e.g., memory transfer via direct memory access (DMA)). Without core I/O abstractions like sockets available to GPU code, GPU programs that access the network must coordinate low-level details among a CPU, GPU and a NIC, for example, managing buffers in weakly consistent GPU memory, or optimizing NIC-to-GPU transfers via peer-to-peer DMAs.

This paper introduces **GPUnet**, a native GPU networking layer that provides a socket abstraction and high-level networking APIs to GPU programs. GPUnet enables individual threads in one GPU to communicate with threads in other GPUs or CPUs via standard and familiar socket interfaces, regardless of whether they are in the same or different machines. Native GPU networking cuts the CPU out of GPU-NIC interactions, simplifying code and increasing performance. It also unifies application compute and I/O logic within the GPU program, providing a simpler programing model. GPUnet uses advanced NIC and GPU hardware capabilities and applies sophisticated code optimizations that yield high application performance equal to or exceeding hand-tuned traditional implementations.

GPUnet is designed to foster GPU adoption in two broad classes of high-throughput data center applications: network servers for back end data processing, e.g., media filtering or face recognition, and scale-out distributed computing systems like MapReduce. While discrete GPUs are broadly used in supercomputing systems, their deployment in data centers has been limited. We blame the added design and implementation complexity of integrating GPUs into complex software systems; consequently, GPUnet's goal is to facilitate such integration.

Three essential characteristics make developing efficient network abstractions for discrete GPUs challenging – massive parallelism, slow access to CPU memory, and low single-thread performance. GPUnet accommodates parallelism at the API level by providing coalesced calls invoked by multiple GPU threads at the same point in data-parallel code. For instance, a GPU program computing a vector sum may receive input arrays from the network by calling `recv()` in thousands of GPU threads. These calls will be coalesced into a single receive request to reduce the processing overhead of the networking stack. GPUnet uses recent hardware support for network transmission directly into/from GPU memory to minimize slow accesses from the GPU to system memory. It provides a reliable stream abstraction with GPU-managed flow control. Finally, GPUnet minimizes control-intensive sequential execution on performance-critical paths by offloading message dispatching to the NIC via remote direct memory access (RDMA) hardware support. The GPUnet prototype supports sockets for network communications over InfiniBand RDMA and supports inter-process communication on a local machine (often called UNIX-domain sockets).

We build a face verification server using the GPUnet prototype that matches images and interacts with `memcached` directly from GPU code, processing 53K client requests/second on a single NVIDIA K20Xm GPU, exceeding the throughput of a 6-core Intel CPU and a CUDA-based server by 1.5× and 2.3× respectively, while maintaining 3× lower latency than the CPU and requiring half as much code than other versions. We also implement a distributed in-GPU-memory MapReduce framework, where GPUs fully control all of the I/O: they read and write files (via GPUfs [35]), and communicate over Infiniband with other GPUs. This architec-

[1] Corresponding author: mark@ee.technion.ac.il

ture demonstrates the ability of GPUnet to support complex communication patterns across GPUs, and for word count and K-means workloads it scales to four GPUs providing speedups of 2.9–3.5× over one GPU.

This paper begins with the motivation for building GPUnet (§2), a review of the GPU and network hardware architecture (§3), and high-level design considerations(§4). It then makes the following contributions:

- It presents for the first time a socket abstraction, API, and semantics suitable for use with general purpose GPU programs (§5).
- It presents several novel optimizations for enabling discrete GPUs to control network traffic (§6).
- It develops three substantial GPU-native network applications: a matrix product server, in-GPU-memory MapReduce, and a face verification server (§7).
- It evaluates GPUnet primitives and entire applications including multiple workloads for each of the three application types (§8).

2. Motivation

GPUs are widely used for accelerating parallel tasks in high-performance computing, and their architecture has been evolving to enable efficient execution of complex, general-purpose workloads. However the use of GPUs in network servers or distributed systems poses significant challenges. The list of 200 popular general-purpose GPU applications recently published by NVIDIA [3] has no mention of GPU-accelerated network services. Using GPUs in software routers and SSL protocols [16, 19, 37], as well as in distributed applications [12] resulted in significant speedups but required heroic development efforts. Recent work shows that GPUs can boost power efficiency and performance for web servers [5], but the GPU prototype lacked an actual network implementation because GPU-native networking support does not yet exist. We believe that enabling GPUs to access network hardware and the networking software stack directly, via familiar network abstractions like sockets, will hasten GPU integration in modern network systems.

GPUs currently require application developers to build complicated CPU-side code to manage access to the host's network. If an input to a GPU task is transferred over the network, for example, the CPU-side code handles system-level I/O issues, such as how to overlap data access with GPU execution and how to optimize the size of memory transfers. The GPU application programmer has to deal with bare-metal hardware issues like setting up *peer-to-peer (P2P) DMA* over the PCIe bus. P2P DMA lets the NIC directly transfer data to and from high-bandwidth graphics double data rate (GDDR) GPU local memory. Direct transfers between the NIC and GPU eliminate redundant PCIe transfers and data copies to system memory, improving data transfer throughput and reducing latency (§8.1). Enjoying the benefits of P2P DMA, however, requires intimate knowledge of hardware-specific APIs and characteristics, such as the underlying PCIe topology.

These issues dramatically complicate the design and implementation of GPU-accelerated networking applications, turning their development into a low-level system programming task. Modern CPU operating systems provide high-level I/O abstractions like sockets, which eliminate or hide this type of programming complexity from ordinary application developers. GPUnet is intended to do the same for GPU programmers.

Consider an internal data center network service for on-demand face-in-a-crowd photo labeling. The algorithm detects faces in the input image, creates face descriptors, fetches the name label for each descriptor from a remote database, and returns the location and the name of each face in the image. This task is a perfect candidate for GPU acceleration because some face recognition algorithms are an order of magnitude faster on GPUs than on a single CPU core [4] and by connecting multiple GPUs, server compute density can be increased even further. Designing such a GPU-based service presents several system-level challenges.

No GPU network control. A GPU cannot initiate network I/O from within a GPU kernel. Using P2P DMA, the NIC can place network packets directly in local GPU memory, but only CPU applications control the NIC and perform send and receive. In the traditional GPU-as-coprocessor programming model, a CPU cannot retrieve partial results from GPU memory while a kernel producing them is still running. Therefore, a programmer needs to wait until all GPU threads terminate in order to request a CPU to invoke network I/O calls. This awkward model effectively forces I/O to occur only on GPU kernel invocation boundaries. In our face recognition example, a CPU program would query the database soon after detecting even a single face, in order to pipeline continued facial processing with database queries. Current GPU programming models make it difficult to achieve this kind of pipelining because GPU kernels must complete before they perform I/O. Thus, all the database queries will be delayed until after the GPU face detection kernel terminates, leading to increased response time.

Complex multi-stage pipelining. Unlike in CPUs, where operating systems use threads and device interrupts to overlap data processing and I/O, GPU code traditionally requires all input to be transferred in full to local GPU memory before processing starts. To overlap data transfers and computations, optimized GPU designs use pipelining: they split inputs and outputs into smaller chunks, and asynchronously invoke the kernel on one chunk, while simultaneously transferring the next input chunk to the GPU, and the prior output chunk from the GPU. While effective for GPU-CPU interaction, the pipeline grows into a complex multi-stage data flow in-

volving GPU-CPU data transfers, GPU invocations and processing of network events. In addition to the associated implementation complexity, achieving high performance requires tedious tuning of buffer sizes which depend on a particular generation of hardware.

Complex network buffer management. If P2P DMA functionality is available, CPU code must set up the GPU-NIC DMA channel by pre-allocating dedicated GPU memory buffers and registering them with the NIC. Unfortunately, these GPU buffers are hard to manage since the network transfers are controlled by a CPU. For example, if the image data exceeds the allocated buffer size, the CPU must allocate and register another GPU buffer (which is slow and may exhaust NIC or GPU hardware resources), or the buffer must be freed by copying the old contents to another GPU memory area. GPU code must be modified to cope with input stored in multiple buffers. While on a CPU, the networking API hides system buffer management details and lets the application determine the buffer size according to its internal logic rather than GPU and NIC hardware constraints.

GPUnet aims to address these challenges. It exposes a single networking abstraction across all system processors and allows using it via a standard, familiar API, thereby simplifying GPU development and facilitating integration of GPUs into complex software systems.

3. Hardware architecture overview

We provide an overview of the GPU software/hardware model, RDMA networking and peer-to-peer (P2P) DMA concepts. We use NVIDIA CUDA terminology because we implement GPUnet on NVIDIA GPUs, but most other GPUs that support the cross-platform OpenCL standard [15] share the same concepts.

3.1 GPU software/hardware model

GPUs are parallel processors that expose programmers to hierarchically structured hardware parallelism (for full details see [23]). They comprise several big cores, *Streaming Multiprocessors* (*SMs*), each having multiple hardware contexts and several Single Instruction, Multiple Data (SIMD) units. All the SMs access global GPU memory and share an address space.

The programming model associates a GPU thread with a single element of a SIMD unit. Threads are grouped into *threadblocks* and all the threads in a threadblock are executed on the same SM. The threads within a threadblock may communicate and share state via on-die shared memory and synchronize efficiently. Synchronization across threadblocks is possible but it is much slower and limited to atomic operations. Therefore, most GPU workloads comprise multiple loosely-coupled tasks each running in a single threadblock, and each parallelized for tightly-coupled parallel execution by the threadblock threads. Once a threadblock has been dispatched to an SM, it cannot be preempted and occupies that SM until all of the threadblock's threads terminate.

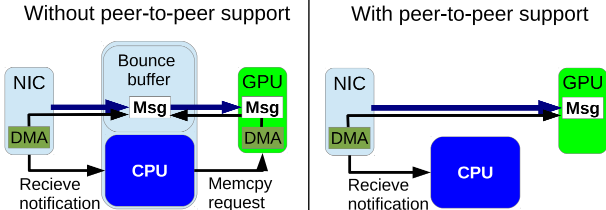


Figure 1: Receiving network messages into a GPU. Without P2P DMA, the CPU must use a GPU DMA engine to transfer data from the CPU bounce buffer.

The primary focus of this work is on discrete GPUs, which are peripheral devices connected to the host system via a standard PCI Express (PCIe) bus. Discrete GPUs feature their own physical memory on the device, with a separate address space that cannot be referenced directly by CPU programs. Moving the data in and out of GPU memory efficiently requires DMA.[2] CPU prepares the data in GPU memory, invokes a GPU *kernel*, and retrieves the results after the kernel terminates.

Interaction with I/O devices. *P2P DMA* refers to the ability of peripheral devices to exchange data on a bus without sending data to a CPU or system memory. Modern discrete GPUs support P2P DMA between GPUs themselves, and between GPUs and other peripheral devices on a PCIe bus, e.g., NICs. For example, the Mellanox Connect-IB network card (HCA) is capable of transferring data directly to/from the GPU memory of NVIDIA K20 GPUs (see Figure 1). P2P DMA improves the throughput and latency of GPU interaction with other peripherals because it eliminates an extra copy to/from bounce buffers in CPU memory, and reduces load on system memory [27, 28].

RDMA and Infiniband. *Remote Direct Memory Access* (RDMA) allows remote peers to read from and write directly into application buffers over the network. Multiple RDMA-capable transports exist, such as Internet Wide Area RDMA Protocol (iWARP), Infiniband and RDMA over Converged Ethernet (RoCE). As network data transfer rates grow, RDMA-capable technologies have been increasingly adopted for in-data center networks, enabling high throughput and low latency networking, surpassing legacy Ethernet performance and cost efficiency [8]. For example, the state-of-the-art fourteen data rate (FDR) Infiniband provides 56Gbps throughput and sub-microsecond latency, with the 40Gbps quad data rate (QDR) technology widely deployed since 2009. Infiniband is broadly used in supercomputing systems and enterprise data centers, and analysts anticipate significant growth in the coming years.

An Infiniband NIC is called a Host Channel Adapter (HCA) and like other RDMA networking hardware, it

[2] NVIDIA CUDA 6.0 provides CPU-GPU software shared memory for automatic data management, but the data transfer costs remain.

performs full network packet processing in hardware, enables zero-copy network transmission to/from application buffers, and bypasses the OS kernel for network API calls.

The HCA efficiently dispatches thousands [18] of network buffers, registered by multiple applications. In combination with P2P DMA, the HCA may access buffers in GPU memory. The low-level *VERB* interface to RDMA is not easy to use. Instead, system software uses VERBs to implement higher-level data transfer abstractions. For example, the *rsockets* [32] library provides a familiar socket API in user-space for RDMA transport. Rsockets are a drop-in replacement for sockets (via LD_PRELOAD), providing a simple way to perform streaming over RDMA.

4. Design considerations

There are many alternative designs for GPU networking; this section discuses important high-level tradeoffs.

4.1 Sockets and alternatives

The GPUnet interface uses sockets because we believe they offer the best blend of properties, being generic, familiar, convenient to use, and versatile (e.g., inter-process communication over UNIX domain sockets). Alternatives like remote direct memory access (RDMA) via a VERBs API are too difficult to program [39]. Existing message passing frameworks (e.g., MPI) [2] allow zero-copy transfers into GPU memory, but they keep all network I/O control on the CPU, and suffer from the conceptual limitations of the GPU-as-slave model that we address in this work.

4.2 Discrete GPUs

We develop GPUnet for discrete GPUs, even though hybrid CPU-GPU processors and system-on-chip options like AMD Kaveri and Qualcomm Snapdragon are gaining market share. We believe discrete and hybrid GPUs will continue to co-exist for years to come. They embody different tradeoffs between power consumption, production costs and system performance, and thus serve different application domains. The aggressive, throughput-optimized hardware designs of discrete GPUs rely heavily on a multi-billion transistor budget, tight integration with specialized high-throughput memory, and increased thermal design power (TDP). Therefore, discrete GPUs outperform hybrid GPUs by an order of magnitude in compute capacity and memory bandwidth, making them attractive for the data center, and therefore a reasonable choice for prototyping GPU networking support.

4.3 Network server organization

Figure 2 depicts different organizations for a multi-threaded network server. In a CPU server (left), a daemon thread accepts connections and transfers the socket to worker threads. In a traditional GPU-accelerated network server (middle) the worker threads invoke computations on a GPU. GPUs are treated as bulk-synchronous high-performance accelerators, so all of the inputs are read on the CPU first and transferred to the GPU across a PCIe bus. This design requires large batches of work to amortize CPU-GPU communications and invocation overheads, which otherwise dominate the execution time. For example, SSLShader [19] needs 1,024 independent network flows on a GTX580 GPU to surpass the performance of 128-bit AES-CBC encryption of a single AES-NI enabled CPU. Batching complicates the implementation, and leads to increased response latency, because GPU code does not communicate with clients directly.

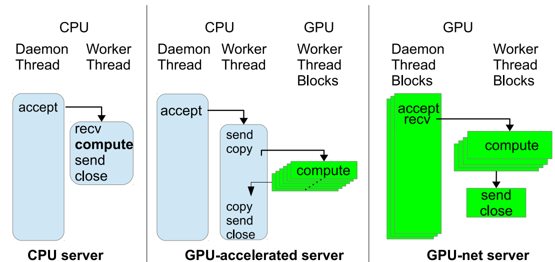


Figure 2: The architecture of a network server on a CPU, using a GPU as a co-processor, and with GPUnet (daemon architecture).

GPUnet makes it possible for GPU servers to handle multiple independent requests without having to batch them first (far right in Figure 2), much like multitasking in multi-core CPUs. We call this the *daemon architecture*. It is also possible to have a GPUnet server where each threadblock acts as an independent server, accepting, computing, and responding to requests. We call this the *independent architecture*. We measure both in §8.

This organization changes the tradeoffs a designer must consider for a networked service because it removes the need to batch work so heavily, thereby greatly simplifying the programming model. We hope this model will make the computational power of GPUs more easily accessible to networked services, but it will require the development of *native GPU programs*.

4.4 In-GPU networking performance benefits

A native GPU networking layer can provide significant performance benefits for building low-latency servers on modern GPUs, because it eliminates the overheads associated with using GPUs as accelerators.

Figure 3 illustrates the flow of a server request on a traditional GPU-accelerated server (top), and compares it to the flow on a server using GPU-native networking support. In-GPU networking eliminates the overheads of CPU-GPU data transfer and kernel invocation, which penalize short requests. For example, computing the matrix product of two 64x64 matrices on a TESLA K20c GPU requires about 14μsec of computation. In comparison, we measure GPU kernel invocation requiring an average of 25μsec and CPU-GPU-CPU data transfers for this size input average 160μsecs.

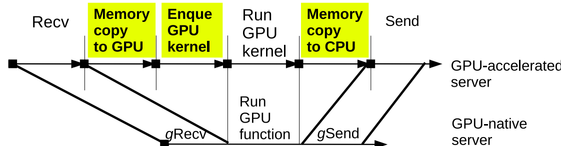


Figure 3: The logical stages for a task processed on a GPU-accelerated CPU server (top) and GPU-native network server(bottom). Highlighted stages are eliminated by the GPU networking support.

In-GPU networking may eliminate the kernel invocation entirely, and provides a convenient interface to network buffers in GPU memory. One potential caveat, however, is that I/O activity on a GPU reduces the GPU's computing capacity, because GPU I/O calls do not relinquish the GPU's resources, as discussed in Section 8.

5. GPUnet Design

Figure 4 shows the high level architecture of GPUnet. GPU programs can access the network via standard socket abstractions provided by the GPUnet library, linked into the application's GPU code. CPU applications may use standard sockets to connect to remote GPU sockets. GPUnet stores network buffers in GPU memory, keeps track of active connections, and manages control flow for their associated network streams. The GPUnet library works with the host OS on the CPU via a GPUnet I/O proxy to coordinate GPU access to the NIC and to the system's network port namespace.

Our goals for GPUnet include the following:

1. **Simplicity.** Enable common network programming practices and provide a standard socket API and an in-order reliable stream abstraction to simplify programming and leverage existing programmer expertise.
2. **Compatibility with GPU programming.** Support common GPU programming idioms like threadblock-based task parallelism and using on-chip scratchpad memory for application buffers.
3. **Compatibility with CPU endpoints.** A GPUnet network endpoint has identical capabilities as a CPU network endpoint, ensuring compatibility between networked services on CPUs and GPUs.
4. **NIC sharing.** Enable all GPUs and CPUs in a host to share the NIC hardware, allowing concurrent use of a NIC by both CPU and GPU programs.
5. **Namespace sharing.** Share a single network namespace (ports, IP addresses, UNIX domain socket names) among CPUs and GPUs in the same machine to ensure backward compatibility and interoperability of CPU- and GPU-based networking code.

5.1 GPU networking API

Socket abstraction. GPUnet sockets are similar to CPU sockets. As in a CPU, a GPU thread may open and use multiple sockets concurrently. GPU sockets are shared across all GPU threads, but cannot be migrated to processes running on other GPUs or CPUs in the same host.

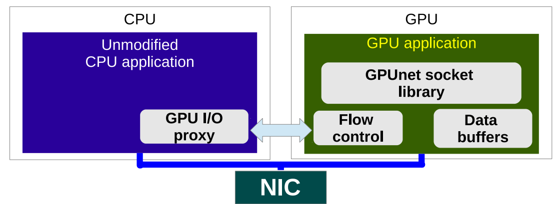


Figure 4: GPUnet high-level design.

GPUnet supports the main calls in the standard network API, including `connect`, `bind`, `listen`, `accept`, `send`, `recv`, `shutdown`, and `close` and their non-blocking versions. In the paper and in the actual implementation we add a "g" prefix to emphasize that the code executes on a GPU. These calls work mostly as expected, though we introduce coalesced multithreaded API calls as we now explain.

Coalesced API calls. A traditional CPU network API is single-threaded, i.e., each thread can make independent API calls and receive independent results. GPU threads, however, behave differently from CPU threads. They are orders of magnitude slower, and the hardware is optimized to run groups of threads (e.g. 32 in an NVIDIA warp or 64 in an AMD wavefront) in lock-step, performing poorly if these threads execute divergent control paths. GPU hardware facilitates collaborative processing inside a threadblock by providing efficient sharing and synchronization primitives for the threads in the same threadblock. GPU programs, therefore, are designed with this hierarchical parallelism in mind: they exploit coarse-grain task parallelism across multiple threadblocks, and process a single task using all the threads in a threadblock jointly, rather than in each thread separately. Performing *data-parallel* API calls in such code is more natural than the traditional per-thread API used in CPU programs. Furthermore, networking primitives tend to be control-flow heavy and often involve large copies between system and user buffers (e.g., `recv` and `send`), making per-threadblock calls superior to per-thread granularity.

GPUnet requires applications to invoke its API at the granularity of a single threadblock. All threads in a threadblock must invoke the same GPUnet call together in a coalesced manner: with the same arguments, at the same point in application code (similar to vectorized I/O calls [42]). These collaborative calls together comprise one logical GPUnet operation. This idea was inspired by a similar design for the GPU file system API [34].

We illustrate coalesced calls in Figure 5. It shows a simple GPU server which increments each received character by one and sends the results back. All GPU threads invoke the same code, but each threadblock executes it independently from others. The threads in a threadblock collaboratively invoke the GPUnet functions to receive/send the data to/from a shared buffer, but perform computations independently in a data-parallel manner. The GPUnet functions are logically executed in lockstep.

```
increment_by_one_server(int csoc)
{
  //buffer shared by all TB threads
   __shared__ char buf[THREADS_IN_TB];
  //collaborative recv into buf
  len=THREADS_IN_TB;
  grecv(csoc, buf, len);
  //data parallel code per thread
  buf[thread_id]++;
  //collaborative send from buf
  gsend(csoc, buf, len);
}
```

Figure 5: A GPU network client using GPUnet (TB – threadblock).

5.2 GPU-NIC interaction

Building a high-performance GPU network stack requires offloading non-trivial packet processing to NIC hardware.

The majority of existing GPU networking projects (with the notable exception of the GASPP packet processing framework [40]) employ the CPU OS network stack with network buffers in CPU memory, and explicit application data movement to and from the GPU. Specifically, accelerated network applications, like SSL protocol offloading [19], cannot operate on raw packets and first require transport-level processing by a CPU. However CPU-GPU memory transfers associated with CPU-side network processing are detrimental to performance as we show in the evaluation.

P2P DMA allows network buffers to reside in GPU memory. However, forwarding all network traffic to a GPU would render the NIC unusable for processes running on a CPU and on other GPUs in the system. Further, since a GPU would receive raw network packets, achieving the goal of providing a reliable in-order socket abstraction would require porting major parts of the CPU network stack to the GPU – a daunting task, which to be efficient requires thousands of packets to be batched in order to hide the overheads of the control-heavy and memory intensive processing involved [40].

To bypass CPU memory, eliminate packet processing, and enable NIC sharing across different processors in the system, we leverage RDMA-capable high-performance NICs. The NIC performs all low-level packet management tasks, assembles application-level messages and stores them directly in application memory, ready to be delivered to an application without additional processing. The NIC can concurrently dispatch messages to multiple buffers and multiple applications, while placing source and destination buffers in both CPU and GPU memory. As a result, multiple CPU and GPU applications can share the NIC without coordinating their access to the hardware for every data transfer.

GPUnet uses both a CPU and a GPU to interact with the NIC. It stores network buffers for GPU applications in GPU memory, and leaves the buffer memory management to the GPU socket layer. The per-connection receive and send queues are also managed by the GPU. On the other hand, the CPU controls the NIC via a standard host driver, keeping the NIC available to all system processors. In particular, GPUnet uses the standard CPU interface to initialize the GPU network buffers and register the GPU memory with the NIC's DMA hardware.

5.3 Socket layer

The GPU socket layer implements a reliable in-order stream abstraction over low-level network buffers and reliable RDMA message delivery. We adopt an RDMA term *channel* to refer to the RDMA connection. The CPU processes all channel creation related requests (e.g., `bind`), allowing GPU network applications to share the OS network name space with CPU applications. Once the channel has been established, however, the CPU steps out of the way, allowing the GPU socket to manage the network buffers as it sees fit.

Mapping streams to channels. GPUnet maps streams one-to-one onto RDMA channels. A channel is a low-level RDMA connection that does not have flow control,[3] so GPUnet must provide flow control using a ring buffer described in Section 6.1. By associating each socket with a channel and its private, fixed-sized send and receive buffers, there is no sharing between streams and hence no costly synchronization. Per-stream channels allows GPUnet to offload message dispatch to the highly scalable NIC hardware. The NIC is capable of maintaining a large number of channels associated with one or more memory buffers.[4]

We considered multiplexing several streams over a single channel, similar to SST [14], which could improve network buffer utilization and increase PCIe throughput due to the increased granularity of memory transfers. We dismissed this design because handling multiple streams over the same channel would require synchronization of concurrent accesses to the same network buffer, which is slow and complicates the implementation.

Naming and address resolution. GPUnet relies on the CPU standard name resolution mechanisms for RDMA transports (CMA) which provide IP-based addressing for RDMA services to initiate the connection.

Wire protocol and congestion control. GPUnet uses reliable RDMA transport services provided by the NIC hardware and therefore relies on the underlying transport packet management and congestion control.

6. Implementation

We implement GPUnet for NVIDIA GPUs and use Mellanox Infiniband Host Channel Adaptors (HCA) for inter-GPU networking [1].

[3] While the Infiniband transport layer does have its own flow control, it is message-oriented and we do not use it for streaming.

[4] Millions for Mellanox Connect-IB, according to Mellanox Solution Brief `http://www.mellanox.com/related-docs/applications/SB_Connect-IB.pdf`

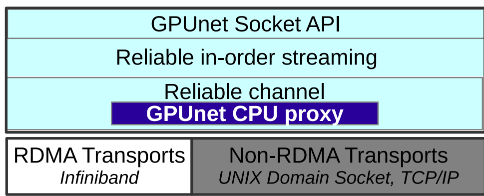


Figure 6: GPUnet network stack.

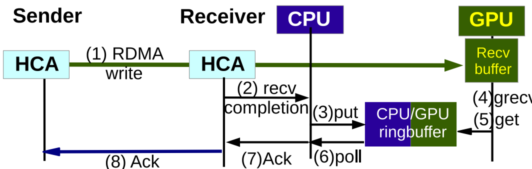


Figure 7: Ring buffer updates for GPU flow control mechanism in `grecv()` call.

GPUnet follows a layered design shown in Figure 6. The lowest layer exposes a *reliable channel* abstraction to upper layers and its implementation depends on the underlying transport. We currently support RDMA, UNIX domain sockets and TCP/IP. The middle *socket layer* implements a reliable in-order connection-based stream abstraction on top of each channel. It manages flow control for the network buffers associated with each connection stream. Finally, the top layer implements the blocking and non-blocking versions of standard socket API for the GPU.

6.1 Socket layer

GPUnet's socket interface is compatible with and builds upon the open-source *rsockets* [32] library for socket-compatible data streams over RDMA for CPUs. Rsockets is a drop-in replacement for sockets (via LD_PRELOAD) which provides a simple way to use RDMA over Infiniband. GPUnet extends the library to use network buffers in GPU memory and integrates it with the GPU flow control mechanisms.

GPUnet maintains a private socket table in GPU. Each active socket is associated with a single reliable channel, and holds the flow control metadata for its receive and send buffers. The primary task of the socket layer is to implement the reliable stream abstraction, which requires flow control management as we describe next.

Flow control. The flow control mechanism allows the sender to block if the receiver's network buffer is full. Therefore, an implementation requires the receiver to update the sender upon buffer consumption.

Unfortunately, our original design to handle flow control entirely on the GPU is not yet practical on current hardware. NVIDIA GPUs cannot yet control an HCA directly, without additional help from a CPU. They cannot access the HCA's "door-bell" registers in order to trigger a send operation, because accessing the door-bell registers is done through memory mapped I/O, and GPUs cannot currently map that memory. Further, the HCA driver does not yet allow placement of completion queue structures in GPU memory. The HCA uses completion queues to deliver completion notifications, e.g., when new data arrives. Therefore, a CPU is necessary to assist every GPU send and receive operation.

Using a CPU for handling completion notifications introduces an interesting challenge for the flow control implementation. The flow control counters must be shared between a CPU and a GPU, since they are updated by a CPU as a part of the completion notification handler, and by a GPU for every `gsend`/`grecv` call. To guarantee consistent concurrent updates, these writes have to be performed atomically, but the updates are performed via a PCIe bus which does not support atomic operations. The solution is to treat the updates as two independent instances of producer-consumer coordination: between a GPU and an HCA (which produces the received data in the GPU network buffer), and between a GPU and a remote host (which consumes the sent data from the GPU network buffer). In both cases, a CPU serves as a mediator for updating the counters in GPU-accessible memory on behalf of the HCA or remote host. Assuming only one consumer and producer, each instance of a producer-consumer coordination can be implemented using a ring-buffer data structure shared between a CPU and a GPU.

Figure 7 shows the ring buffer processing a receive call. The GPU receives the data into the local buffer via direct RDMA memory copy from the remote host (1). The CPU gets notified by the HCA that the data was received (2) and updates the ring buffer as a producer on behalf of the remote host (3). Later, the GPU calls `grecv()` (4), reads the data and updates the ring buffer that the data has been consumed (5). This update triggers the CPU (6) to send a notification (7) to the remote host (8).

This design decouples the GPU API calls and the CPU I/O transfer operations, allowing the CPU to handle GPU I/O request asynchronously. As a result, the GPU I/O call returns faster, without waiting for the GPU I/O request to propagate through the high-latency PCIe bus, and data transfers and GPU computations are overlapped. This feature is essential to achieve high performance for bulk transfers.

6.2 Channel layer

The channel layer mediates the GPU's access to the underlying network transport and runs on both CPU and GPU. On the GPU side it manages the network buffers in GPU memory, while the CPU side logic ensures that the buffers are delivered to and from the transport mechanism underneath, as we describe shortly.

Memory management. GPUnet allocates a large contiguous region of GPU memory which it uses for network buffers. To enable RDMA hardware transport, the CPU code registers the GPU memory into the Infiniband HCA with the help of CUDA's GPUDirectRDMA mechanism. The maximum total amount of HCA registered memory

is limited to 220MB in NVIDIA TESLA K20c GPUs due to the Base Address Register (BAR) size constraints of the current hardware. We allocate the memory statically during GPUnet initialization because the memory registration is expensive, and also because we were unable to register it while the GPU kernel is running. GPUnet uses this RDMA-registered memory as a memory pool for allocating a receive and send buffer for each channel.

Bounce buffers and support for non-RDMA transports. If P2P DMA functionality is not available, the underlying transport mechanism has no direct access to GPU network buffers. Therefore, network data must be explicitly staged to and from *bounce buffers* in CPU memory.

Using bounce buffers has higher latency and requires larger system buffer than native RDMA, as we measure in Section 8.1. However, this functionality serves to bridge current hardware constraints, which often make the use of RDMA impossible or inefficient. P2P DMA for GPUs and other peripherals has been made available only since early 2013, and its hardware and software support is still immature. For example, on some modern server chipsets we encountered $15\times$ bandwidth degradation when storing send buffers in GPU memory, and as a result had to use bounce buffers. Similarly, P2P DMA is only possible in a certain PCIe topology, so for our dual socket configuration only one of the three PCIe attached GPUs can perform P2P DMA with the Infiniband HCA. Until the software and hardware support stabilizes, bounce buffers are an interim solution that hides the implementation complexity of CPU-GPU-NIC coordination mechanisms.

6.3 Performance optimizations.

Single threadblock I/O. While developing GPUnet applications we found that it is convenient to dedicate some threadblocks to performing network operations, while using others only for computation, like the receiving threadblock in MapReduce (§7.1), or a daemon threadblock in the matrix product server (§7). In such a design, the performance-limiting factor for send operations is the *latency* of two steps performed in the `send()` call: memory copy between the system and user buffers in GPU, and the update of the flow control ring buffer metadata.

Unfortunately, a single threadblock is allocated only a small fraction of the total GPU compute and memory bandwidth resources, e.g. up to 7% of the total GPU memory bandwidth according to our measurements. Improving the memory throughput of a single threadblock requires issuing many memory requests per thread in order to enable memory-level parallelism [41]. We resorted to PTX, NVIDIA GPU low-level assembly, in order to implement 128-bit/thread vector accesses to global memory which also bypass the L2 and L1 caches. This bypassing is required to ensure a consistent buffer state when RDMA operations access GPU memory. This optimization improves memory copy throughput almost $3\times$, from 2.5GB/s to 6.9GB/s for a threadblock with only 256 threads.

Ring buffer updates. Ring buffer updates were slow initially because the metadata is shared between the CPU and GPU, and we placed it in "zero-copy" memory, which physically resides on a CPU. Therefore, reading this memory from the GPU incurs a significant penalty of about 1-2μsec. Updating the ring buffer requires multiple reads, and the latency accumulates to tens of μsec.

We improved the performance of ring buffer updates by converting reads from remote memory into remote writes into local memory. For example, the head location of a ring buffer, which is updated by a producer, should reside in the consumer's memory in order to enable the consumer to read the head quickly. To implement this optimization, however, we must map GPU memory into the CPU's address space, which is not supported by CUDA. We implement our own mapping using NVIDIA's GPUDirect from a Linux kernel module. This optimization reduces the latency of ring buffer updates to 2.5μsec.

6.4 Limitations

GPUnet does not provide a mechanism for socket migration between a GPU and a CPU, which might be convenient for load balancing.

More significantly, the prototype relies on the ability of a GPU to provide the means to guarantee consistent reads to its memory when it is concurrently accessed by a running kernel and the NIC RDMA hardware. NVIDIA GPUs do not currently provide such consistency guarantees. In practice, however, we do not observe consistency violations in GPUnet. Specifically, to validate our current implementation, we implement a GPU CRC32C library and instrument the applications to check the data integrity of all network messages with 4KB granularity. We detect no data integrity violations for experiments reported in the paper (though this experiment surfaced a small bug in GPUnet itself).

We hope, perhaps encouraged by GPUnet itself, that GPU vendors will provide such consistency guarantees in the near future. In fact, the necessary CPU-GPU memory consistency will be supported in the future releases of OpenCL 2.0-compliant GPU platforms, thereby supporting our expectation that it will become the standard guarantee in future systems.

7. Applications

Matrix product server. The matrix product server is implemented entirely on the GPU, using both the daemon and independent architectures (§4.3). In the daemon architecture the daemon threadblock (one or more) accepts a client connection, reads the input matrices, and enqueues a multiplication kernel. The multiplication kernel gets pointers to the input matrices and the socket for writing the results. The number of threads – a critical param-

eter defining how many GPU computational resources a kernel should use – is derived from the matrix dimensions as in the standard GPU implementation. When the execution completes, the threadblock which finalizes the computation sends the data back to the client and closes the connection.

In the independent architecture each threadblock receives the input, runs the computations, and sends the results back.

Implementation details. The daemon server cannot invoke the multiplication kernel using dynamic parallelism (which is the ability to execute a GPU kernel from within an executing kernel, present since NVIDIA Kepler GPUs). Current dynamic parallelism support in NVIDIA GPUs lacks a parent-child concurrency guarantee, and in practice the parent threadblock blocks to ensure the child starts its execution. Our daemon threadblock must remain active to accept new connections and handle incoming data, so we do not use NVIDIA's dynamic parallelism and instead invoke new GPU kernels via CPU by a custom mechanism. See Section 8.2 for performance measurements.

7.1 MapReduce design

We design an in-GPU-memory distributed MapReduce framework that keeps intermediate results of map operation in GPU memory, while input and output are read from disk using GPUfs [34]. We call the system *GimMR* for *G*PU *i*n *m*emory *M*ap *R*educe. GimMR is a native GPU application without CPU code. The number of GPUs in our system is small, so all of them are used to execute both mappers and reducers. Shuffling (i.e., the exchange of intermediate data produced by mappers between different hosts) is done by mappers, and reducers only start once all mappers and data transfer has completed. Our mappers push data, while in traditional MapReduce, the reducers pull [13]. Each GPU runs multiple mappers and reducers, each of which are executed by multiple GPU threads.

At the start of the Map phase a mapper reads its part of the input via GPUfs. The input is split across all threadblocks, so they can execute in parallel. A GPU may run tens of mappers, each with hundreds of threads. Mappers generate intermediate <key,value> pairs that they assign to *bucket*s using consistent hashing or a predefined key range. Buckets contain pointers to data chunks. A mapper accumulates intermediate keys and data into local chunks. When a chunk size exceeds a threshold, the mapper sends the chunk to the GPU which will run the reducer for the keys in that bucket, thereby overlapping mapper execution with the shuffle phase, similar to ThemisMR [29].

Each Map function is invoked in one threadblock and is executed by all the threadblock threads. On each GPU, there are many mapper threadblocks and consumer threadblocks, with the consumer threadblocks receiving buckets from remote GPUs. Each consumer threadblock is assigned a fixed number of connections from a remote GPU. The receivers get data by making non-blocking calls to `grecv()` on the mappers' sockets in round-robin order (using `poll()` on the GPU is left as future work).

The network connections are set up at the beginning of the Map phase, between each pair of consumer threadblock and remote threadblock. For example, a GPU node in a GimMR system with five GPUs, each with 12 mapper and 12 consumer threadblocks, will have a total of 48 incoming connections, one per mapper from every other GPU. And each of its 12 consumers will handle 4 incoming connections. Local mappers update local buckets without sending them through the network.

GPU mappers coordinate with a CPU-side centralized mapper master, accessed over the network. The master assigns jobs, balancing load across the mappers. The master tells each mapper the offset and size of the data it should read from its input file.

Similar to the Map, each Reduce function is also invoked in one threadblock. Each reducer identifies the set of buckets it must process, (optionally) performs parallel sort of all the key-value pairs in each bucket separately, and finally invokes the user-provided Reduce function. As a result, the GPU exploits the standard coarse-grain data parallelism of independent input keys, but also enables the finer-grained parallelism of a function processing values from the same key, e.g., by parallel sorting or reduction. Enabling each reducer to sort the key/values independently of other reducers is important to avoid a GPU-wide synchronization phase at the end of sorting.

GimMR takes advantage of the dynamic communication capabilities of GPUnet for ease and efficiency in implementation. Without GPUnet, enabling overlapped communications and computations would require significant development effort involving fine-tuned pipelining among CPU sends, CPU-GPU data transfers, and GPU kernel invocations.

GimMR workloads. We implement word count and K-means. In word count, the mapper parses free-form input text and generates <word, 1> pairs, which are reduced by summing up their values. CUDA does not provide text processing functions, so we implement our own parser. We pre-sample the input text and determine the range of keys being reduced by each reducer.

The mappers in K-means calculate the distance of each point to the cluster centroids, and then re-cluster the point to its nearest centroid. Intermediate data is pairs of <centroid number, point>. The reducer sums the coordinates of all points in a centroid. K-means is an iterative algorithm, and our framework supports iterative MapReduce. A CPU process receives the results of the reducers, and calculates the new centroids for the next round.We preprocess the input file to piecewise transpose the input

points, thereby coalescing memory accesses for threads in a threadblock.

7.2 Face verification

A client sends a photo of a face, along with a text label identifying the face, to a verification service. The server responds positively if the label matches the photo (i.e., the server has the same face in its database with the proffered label), and negatively otherwise. The server uses a well-known local binary patterns (LBP) algorithm for face verification [6]. LBP represents images by a histogram of their visual features. The server stores all LBP histograms in a `memcached` database. In our testbed, we have three machines, one for clients, one for the verification server and one for the `memcached` database.

We believe our organization is a reasonable choice, as opposed to alternatives such as having the client perform the LBP and send a histogram to the server. Face verification algorithms are constantly evolving, and placing them on the server makes upgrading the algorithm possible. Also, sending actual pictures to the server provides a useful human-checkable log of activity.

Client. The client uses multiple threads, each running on its own CPU, and maintaining multiple persistent non-blocking connections with the server. Clients use rsockets for network communications with the server. For each connection, the client performs the following steps and repeats them forever:

1. Read a (random) 136x136 grayscale image from a (cached) file.
2. Choose a (random) face label.
3. Send verification request to server.
4. Receive response from server – 0 (mismatch) or 1 (match).

Server. We implement three versions of the server: a CPU version, a CUDA version, and a GPUnet version. Each server performs the following steps repeatedly (in different ways).

1. Receive request from client.
2. Fetch LBP histogram for client-provided name from the remote memcached database.
3. Calculate LBP histogram of the image in the request.
4. Calculate Euclidean distance between the histograms.
5. Report a match if the distance is below a threshold.
6. Send integer response.

The CPU server consists of multiple independent threads, one per CPU core. Each thread manages multiple, persistent, non-blocking connections with the client.

The CUDA server is the same as the CPU server, but the face verification algorithm executes on the GPU by launching a kernel. (see Figure 2, middle picture).

The GPUnet server is a native GPU-only application using GPUnet for network operations. It uses the independent architecture (§4.3), and consists of multiple threadblocks running forever, with each acting as an independent server. Each threadblock manages persistent connections with the client and `memcached` server. This design is appropriate since the processing time per image is low and there is enough parallelism per request.

Node	Chipset Intel	CPU Intel	GPU NVIDIA	DMA	Software
A B	Z87	E3-1220V3 Haswell	K20c	N	RHEL 6.5, gcc 4.4.7, GPU driver 331.38
C	C602	E5-2620 Sandy Bridge	C2075	Y	RHEL 6.3, gcc 4.4.6, GPU driver 319.37
D	5520	2× L5630 Westmere	2× C2075	Y	RHEL 6.3, gcc 4.4.6, GPU driver 319.37

Table 1: Hardware and software configuration. The DMA column indicates the presence of a DMA performance asymmetry (§6.2).

Implementation details. We use a standard benchmarking face recognition dataset[5], resized to 136x136 and reformatted as raw grayscale images. We implement a GPU `memcached` client library. `memcached` uses Infiniband RDMA transport provided by the rsockets library. We modified a single line of `memcached` to work with rsockets by disabling the use of `accept4`, which is not supported by rsockets.

8. Evaluation

Hardware. We run our experiments on a cluster with four nodes (Table 1) connected by a QDR 40Gbps Infiniband interconnect, using Mellanox HCA cards with MT4099 and MT26428 chipsets.

All machines use CUDA 5.5. ECC on GPUs, hyperthreading, SpeedStep, and Turbo mode of all the machines are disabled for reproducible performance. Nodes A and B feature a newer chipset with a PLX 8747 PCIe switch which enables full bandwidth P2P DMA between the HCA and the GPU. Nodes C and D provide full bandwidth for DMA writes from HCA to GPU (`grecv()`), but perform poorly with only 10% of the bandwidth for DMA reads from GPU (`gsend()`). We are not the first to observe such asymmetry [28].

GPUnet delegates connection establishment and teardown to a CPU. Our benchmarks exclude connection establishment from the performance measurement to measure the steady-state behavior of persistent connections. Using persistent connections is a common optimization technique for data center applications [11].

8.1 Microbenchmarks

We run microbenchmarks with two complementary goals: to understand the performance consequences of GPUnet design decisions, and to separate the essential bottlenecks from the ephemeral issues due to current hardware. We run them between nodes A and B with 256 threads per threadblock. All results are the average of 10 iterations, with the standard deviation within 1.1% of the mean.

[5] `http://www.itl.nist.gov/iad/humanid/feret/feret_master.htm`

	C-C	C-G RDMA	C-G BB	G-G RDMA	G-G BB
RTT 64 byte(μsec)	2.86	26.9	60.3	50.0	117
Bandwidth (GB/s)	3.44	3.44	3.48	3.38	3.46

Table 2: Single stream latency (round trip time) and bandwidth for GPUnet, CPU uses rsockets. C–CPU, G-GPU, BB–bounce buffer.

Steps	Latency (μsec)
T_1 GPU ring buffer	1.4
T_2 GPU copies buffer	15.7
T_3 GPU requests to CPU	3.8
T_4 CPU reads GPU request	2.5
T_5 CPU RDMA write time to completion	22.2
Total one-way latency	45.6

Table 3: Latency breakdown for a GPU `gsend()` request with a 64KB message with peer-to-peer RDMA.

Single stream performance. We run a simple single-threadblock GPU echo server and client using a single GPUnet socket. We implement the CPU version of the benchmark using the unmodified rsockets library. Table 2 shows the round trip time (RTT) for 64 byte messages and bandwidth for 64KB messages and 256KB (512KB for bounce buffer) system buffers. The GPU reaches about 98% of the peak performance of CPU-based rsockets. Bounce buffers (entries marked BB in the table) increase latency two-fold versus RDMA transfers, but its throughput is close to RDMA thanks to twice larger system buffers for better latency hiding.

The latency of GPU transfers is significantly higher than the baseline CPU-to-CPU latency. To understand the reasons, Table 3 provides the breakdown for the latency of individual steps of `gsend()` sending 64KB.

We measured T_1, T_2, T_3 on the GPU by instrumenting the GPU code using `clock64()`, the GPU intrinsic that reads the hardware cycle counter. T_5 is effectively the latency of the `send()` call performed from the CPU, but transferring data between memories of two GPUs. For this data size, the overhead of GPU-related processing is about 50%. The user-to-system buffer copy, T_2, is the primary bottleneck. Accessing CPU-GPU shared data structures (T_1, T_3) and the latency of the update propagation through the PCIe bus (T_4) account for 20% of the total latency, but these are constant factors.

We believe that T_2 and T_4 will improve in future hardware generations. Specifically, T_4 can be reduced by enabling a GPU to access the HCA doorbell registers directly, without CPU mediation. We believe that T_2 can be optimized by exposing the already existing GPU DMA engine for performing internal GPU DMAs, similar to the Intel I/OAT DMA engine. Alternatively, a zero-copy API may help eliminate T_2 in software.

Duplex performance. The CPU rsocket library achieves 6.65 GB/s of the aggregate duplex bandwidth for two concurrent data streams in opposite directions – twice the bandwidth of a single stream. With GPUnet, we found that `gsend` and `grecv` interfere when invoked concurrently on two sockets, but the reasons for this interference is still unclear. Specifically, when using a CPU end-point, the throughput of `grecv` and `gsend` is 3.31 GB/s and 2.63 GB/s respectively. As a result, in a GPU-GPU experiment with two opposite streams, the one-directional bandwidth is constrained by the `gsend` performance on both sides, hence the aggregate bandwidth is 5.26 GB/s.

Multistream bandwidth. We measured the aggregate bandwidth of sending over multiple sockets from one GPU. We run 26 threadblocks (2 threadblocks per GPU SM core) each having multiple non-blocking sockets. Each send is 32KB. We test up to 416 active connections – the maximum number of sockets that GPUnet may concurrently maintain given 256KB send buffers, which provide the highest single-stream performance. As we explained in § 6, the maximum number of sockets is constrained by the total amount of RDMA-registered memory available for network buffers, which is currently limited to 220MB.

We run the experiment between two GPUs. Starting from 2 connections, GPUnet achieves a throughput of 3.4GB/s, and gradually falls to 3.2GB/s at 416 connections, primarily due to the increased load on the CPU-side proxy having to handle more requests. Using bounce buffers shows slightly better throughput, 3.5GB/s with two connections, and 3.3GB/s with 208 connections.

8.2 Matrix product server

We implement three versions of the matrix product server to examine the performance of different GPU server organizations.

The *CUDA* server runs the I/O logic on the CPU and offloads matrix product computations to the GPU using standard CUDA. It executes a single CPU thread and invokes one GPU kernel per request (`matrixMul`), the matrix product kernel distributed with the NVIDIA SDK.

The *daemon* server uses GPUnet and follows the daemon architecture (§4.3). The GPU resources are partitioned between daemon threadblocks and computing threadblocks. The number of daemon threadblocks is an important server configuration parameter as we discuss below. Both the CUDA server and the daemon server invoke the matrix product kernel via the CPU, however the latter receives/sends data directly to/from GPU memory.

The *independent* server also employs GPUnet, but the GPU is not statically partitioned between daemon and compute threadblocks. Instead, all the threadblocks handle I/O and perform computations, and no additional GPU kernels are launched.

The CUDA, daemon and independent server versions are 894, 391 and 220 LOC for their core functionality.

Resource allocation in the daemon server. The performance of the daemon server is particularly sensitive to the way GPU resources are partitioned between I/O and compute tasks performed by the server. The GPU non-preemptive scheduling model implies that GPU resources allocated to I/O tasks cannot execute computations even

Configuration	Workload Light	Medium	Heavy
Light	92%	81%	74%
Medium	44%	99%	88%
Heavy	12%	44%	100%

Table 4: The cost of misconfiguration: the throughput in a given configuration relative to the maximum throughput using the best configuration for that workload.

while I/O tasks are idle waiting for the input data. Therefore, if the server is configured to run too many daemon threadblocks, the compute kernels will get fewer GPU resources and computations will execute slowly. On the other hand, too few daemon threadblocks may fail to feed the execution units with data fast enough, thereby decreasing the server throughput [6]. In our current implementation the number of daemon threadblocks is configured at server invocation time and does not change during execution.

The best server configuration depends on the workload. Intuitively, the more computation that is performed per byte of I/O, the fewer GPU resources should be allocated for I/O threadblocks and, consequently, more resources allocated for computation. The optimal server configuration depends on the compute-to-I/O ratio of its tasks.

Balancing the allocation of threadblocks between computation and I/O is a high-stakes game. Table 4 shows how we separate three matrix multiplication workloads by their compute-to-I/O ratio: light (64x64 and 128x128), medium (256x256) and heavy (512x512 and 1024x1024).

We exhaustively search the configuration space for each workload (with varying number of clients) to find the configuration of compute and I/O threadblocks that maximizes throughput. Then we run all workloads on all configurations and measure the penalty for using the best configuration for each class of workload. Splitting workloads into three classes allows us to find configurations that perform very well for all instances of that class (the diagonal is all above 90% of optimal). However, dedicating too many or too few threadblocks to I/O can be terrible for performance, with the worst misconfiguration reducing throughput to 12% of optimal. Future work includes a generic method of finding the best server configuration and dynamically adjusting it to suit the workload.

Performance comparison of different server designs. We compare the throughput of different server designs while changing the number of concurrent clients. We use the 256×256 matrices for input, and configure the daemon server to have the number of daemon threadblocks

[6] In practice, the number of threads per a daemon threadblock also affects the server performance, but we omit these technical details for simplicity.

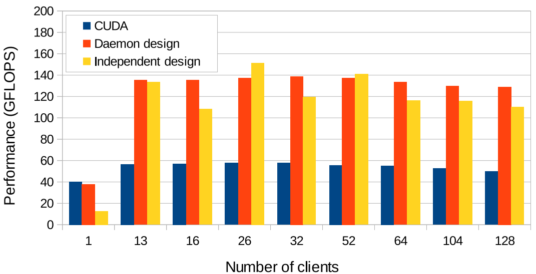


Figure 8: Throughput comparison for different matrix product servers.

Server design	Light workload	Medium workload	Heavy workload
Daemon (GFLOPS)	11	137	201
Independent (GFLOPS)	37 (3.4×)	151 (1.1×)	207 (1.01×)

Table 5: The throughput of GPUnet-based matrix product servers under different workload types.

which maximizes its throughput for this workload. The results are shown in Figure 8.

Both GPUnet-based implementations consistently outperform the traditional CUDA server across all the workloads and are competitive with each other.

As expected, the performance of the independent design is sensitive to the number of clients. Our implementation assigns one connection per threadblock, so the number of clients equals the number of server threadblocks. Configurations where the number of clients are divisible by the number of GPU SMs (13 in our case) have the best performance. Other cases suffer from load imbalance. The performance of the independent design is particularly low for one client because the server runs with a single threadblock using a single SM, leading to severe underutilization of GPU resources.

The performance of the independent design is $8\times$ to $20\times$ higher than a single-threaded CPU-only server that uses the highly-optimized BLAS library (not shown in the figure).

Table 5 shows the throughput of the GPUnet servers serving different workload types. We fixed the number of active connections to 26 to allow the independent server to reach its full performance potential.

The independent server achieves higher throughput for all of the workload types, but its advantages are most profound for light tasks (with low compute-to-I/O ratios). The independent server does not incur the overhead of GPU kernel invocations, which dominate the execution time for shorter tasks in the daemon server. This performance advantage makes the independent design particularly suitable for our face verification server which also runs tasks with low compute-to-I/O ratio as we describe below (§ 8.4).

8.3 Map reduce

We evaluate the standard word count and K-means tasks on our GimMR MapReduce. Table 6 compares the performance of the single-GPU GimMR with the single-node Hadoop and Phoenix++ [38] on a 8-core CPU. We

Workload	8-core Phoenix++	1-Node Hadoop	1-GPU GimMR
K-means	12.2 sec	71.0 sec	5.6 sec
Wordcount	6.23 sec	211.0 sec	29.6 sec

Table 6: Single-node GimMR vs. other MapReduce systems.

use RAM disk and IP over IB when evaluating K-Means on Hadoop. For both wordcount and kmeans on Hadoop, we run 8 map jobs and 16 reduce jobs per node.

Word count. The word count serves as a feasibility proof for distributed GPU-only MapReduce, but the workload characteristics make it inefficient on GPUs.

The benchmark counts words in a 600MB corpus of English-language Wikipedia in XML format. A single GPU GimMR outperforms the single-node 8-core Hadoop by a factor of 7.1×, but is 4.7× slower than Phoenix++ [38] running on 8 CPU cores. GimMR word count spends a lot of time sorting strings, which is expensive on GPUs because comparing variable length strings create divergent, irregular computations. In the future we will adopt the optimization done by ThemisMR [29] which uses the hash of the strings as the intermediate keys, in order to sort quickly.

Scalability. When invoked on the same input on four network-connected GPUs, GimMR performance increases by 2.9×. The scalability is affected by three factors: (1) the amount of computation is too low to fully hide the intermediate data transfer overheads, (2) reducers experience imbalance due to the input data skew, (3) Only two machines enable GPU-NIC RDMA, the other two use bounce buffers.

K-means. We chose K-means to evaluate GimMR under a computationally-intensive workload. We compute 500 clusters on a randomly generated 500MB input with 64K vectors each with hundreds of floating point elements.

Table 6 compares the performance of GimMR with single-node Hadoop and Phoenix++ using 200 dimension vectors. GimMR on a single GPU outperforms Phoenix++ on 8 CPU cores by up to 2.2×, and Hadoop by 12.7×.

Scalability. When invoked on the same input on four network-connected GPUs, GimMR performance increases by 2.9×. With 100 dimension vectors, the 4-GPU GimMR achieves up to 3.5× speedup over a single GPU.

8.4 Face verification

We evaluate the face verification server on a different cluster with three nodes, each with Mellanox Connect-IB HCA, 2× Intel E5-2620 6-core CPU, and connected via a Mellanox Switch-X bridge. The server executes on NVIDIA K20Xm GPUs. The application's client, server and `memcached` server run on their own dedicated machines. We verified that both the CPU and GPU algorithm implementations produce the same results, and also manually inspected the output using the standard FERET

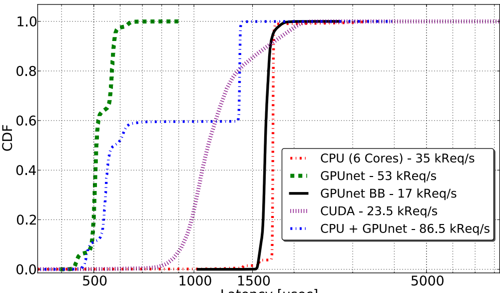


Figure 9: Face verification latency CDF for different servers.

dataset and hand-modified images. All the reported results have variance below 0.1% of their mean.

Lower latency, higher throughput. Figure 9 shows the CDF of the request latency for different server implementations and some of their combinations. The legend for each server specifies the effective server throughput observed during the latency measurements. GPUnet and CUDA are invoked with 28 threadblocks, 1024 threads per threadblock, which we found to provide the best tradeoff between latency and throughput. Other configurations result in higher throughput but sacrifice latency, or slightly lower latency but much lower throughput.

The GPUnet server has the lowest average response time of 524±41 μsec per request while handling 53 KRequests/sec, which is about 3× faster per request, and 50% more requests than the CPU server running on a single 6-core CPU. The native CUDA version and GPUnet with bounce buffers suffer from 2× and 3× higher response time, and 2.3× and 3× lower throughput respectively. They both perform extra memory copies, and the CUDA server is further penalized for invoking a kernel per request. Dynamic kernel invocation accounts for the greater variability in the response time of the CUDA server. The combination of CPU and GPUnet achieves the highest throughput, and improves the server response time for all requests, not only for those served on a GPU.

Maximum throughput and multi-GPU scalability. The throughput-optimized configuration for the GPUnet server differs from its latency-optimized version, with 4× more threadblocks, each with 4× fewer threads (112 threadblocks, each with 256 threads). While the total number of threads remains the same, this configuration serves 4× more concurrent requests. With 4× fewer threads processing each request, the processing time grows only by about 3×. Therefore this configuration achieves about 30% higher throughput as shown in Table 7, which is within 3% of the performance of two 2×6-core CPUs.

Adding another GPU to the system almost doubles the server throughput. Achieving linear scalability, however, requires adding a second Infiniband card. The PCIe topology on the server allows only one of the two GPUs to use P2P DMA with the same HCA, and the second GPU has to fall back to using bounce buffers, which has inferior performance in this case. To work around the

Server type	CPU	2× CPU	CUDA	GPUnet BB	GPUnet	2×GPUnet	2×GPUnet + CPU
Thpt (Req/s)	35K	69K	23K	17K	67K	136K	188K

Table 7: Face verification throughput for different servers.

problem, we added a second HCA to enable P2P DMA for the second GPU.

Finally, invoking both the CPU and GPUnet servers together results in the highest throughput. Because each GPU in GPUnet requires one CPU core to run, the CPU server gets two fewer cores than the standalone CPU version, and the final throughput is lower than the sum of the individual throughputs. The total server throughput is about 172% higher than the throughput of a 6x2-core CPU-only server.

The GPUnet-based server I/O rate with a single GPU reaches nearly 1.1GB/s. I/O activity accounts for about 40% of the server runtime. GPUnet enables high performance with a relatively modest development complexity compared to other servers. The CUDA server has 596 LOC, CPU - 506, and GPUnet– only 245 lines of code.

9. Related work

GPUnet is the first system to provide native networking abstractions for GPUs. This work emerges from a broader trend to integrate GPUs more cleanly with operating system services, as exemplified by recent work on a file system layer for GPUs (GPUfs) [34] and virtual memory management (RSVM [20]).

OS services for GPU applications. GPU applications operate outside of the resource management scope of the operating system, often to the detriment of system performance. PTask [30] proposes a data flow programming model for GPUs that enables the OS to provide fairness and performance isolation. TimeGraph [22] allows a device driver to schedule GPU processors to support real-time workloads.

OSes for heterogeneous architecture. Barrelfish [9] proposes multikernels for heterogeneous systems based on memory decoupled message passing. K2 [25] shows the effectiveness of tailoring a mature OS to the details of a heterogeneous architecture. GPUnet demonstrates how to bring system services into a heterogeneous system.

GPUs for network acceleration. There have been several projects targeting acceleration of network applications on GPUs. For example, PacketShader [16] and Snap [37] use GPUs to accelerate packet routing at wire speed, while SSLShader [19] offloads SSL computations. Numerous high-performance computing applications (e.g., Deep Neural Network learning [12]) use GPUs to achieve high per-node performance in distributed applications. These works use GPUs as co-processors, and do not provide networking support for GPUs. GASPP [40] accelerates stateful packet processing on GPUs, but it is not suitable for building client/server applications.

Peer-to-peer DMA. P2P DMA is an emerging technology, and published results comport with the performance problems GPUnet has on all but the very latest hardware. Potluri et. al. [27, 28] use P2P DMA for NVIDIA GPUs and Intel MICs in an MPI library, and report much less bandwidth with P2P DMA than communication through CPU. Kato et. al [21] and APEnet+ [7] also propose low-latency networking systems with GPUDirect RDMA, but report hardware limitations to their achieved bandwidth. Trivedi et al. [39] point out the limitation of RDMA with its complicated interaction with various hardware components and the effect of architectural limits on RDMA.

Network stack on accelerators. Intel Xeon Phi is a co-processor akin to a GPU, but featuring x86 compatible cores and running embedded Linux. Xeon Phi enables direct access to the HCA from the co-processor and runs a complete network stack [45]. GPUnet provides a similar functionality for GPUs, and naturally shares some design concepts, like the CPU-side proxy service. However, GPUs and Xeon Phi have fundamental differences, e.g. fine-grain data parallel programming model, and the lack of hardware support for operating system, which warrant different approaches to key design components such as the coalesced API and the CPU-GPU coordination.

Scalability on heterogeneous architecture. Dandelion [31] is a language and system support for data-parallel applications on heterogeneous architectures. It provides a familiar language interface to programmers, insulating them from the heterogeneity.

GPMR [36] is a distributed MapReduce system for GPUs, which uses MPI over Infiniband for networking. However, it uses both CPUs and GPUs depending on the characteristics of the steps of the MapReduce.

Network server design. Scalable network server design has been heavily researched as processor and networking architecture advance [10, 17, 24, 33, 43, 44], but most of this work is specific to CPUs.

Rhythm [5] is one of the few GPU-based server architectures that use GPUs to run PHP web services. It promises throughput and energy efficiency that can exceed CPU-based servers, but its current prototype lacks the in-GPU networking that GPUnet provides.

Low-latency networking. More networked applications are demanding low-latency networking. RAMCloud [26] notes the high latency of conventional Ethernet as a major source of latency for a RAM-based server, and discusses RDMA as an alternative that is difficult to use directly.

10. Acknowledgments

Mark Silberstein was supported by the Israel Science Foundation (grant No. 1138/14) and the Israeli Ministry of Science. We also gratefully acknowledge funding from NSF grants CNS-1017785 and CCF-1333594.

References

[1] *GPUnet project web page*. https://sites.google.com/site/silbersteinmark/GPUnet.

[2] MVAPICH2: High performance MPI over InfiniBand, iWARP and RoCE. http://mvapich.cse.ohio-state.edu.

[3] *Popular GPU-accelerated applications*. http://www.nvidia.com/object/gpu-applications.html.

[4] Efficient Object Detection on GPUs using MB-LBP features and Random Forests. GPU Technology Conference, 2013. http://on-demand.gputechconf.com/gtc/2013/presentations/S3297-Efficient-Object-Detection-GPU-\MB-LBP-Forest.pdf.

[5] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[6] T. Ahonen, A. Hadid, and M. Pietikainen. Face description with local binary patterns: Application to face recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(12):2037–2041, 2006.

[7] R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, P. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini. APEnet+: a 3D Torus network optimized for GPU-based HPC Systems. In *Journal of Physics: Conference Series*, volume 396. IOP Publishing, 2012.

[8] T. G. T. analysts. InfiniBand data center march, 2012. https://cw.infinibandta.org/document/dl/7269.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 29–44. ACM, 2009.

[10] N. Z. Beckmann, C. Gruenwald III, C. R. Johnson, H. Kasture, F. Sironi, A. Agarwal, M. F. Kaashoek, and N. Zeldovich. PIKA: A network service for multikernel operating systems. Technical Report MIT-CSAIL-TR-2014-002, MIT, January 2014.

[11] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 267–280. ACM, 2010.

[12] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1337–1345, 2013.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[14] B. Ford. Structured streams: A new transport abstraction. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 361–372, New York, NY, USA, 2007. ACM.

[15] K. Group. *OpenCL - the open standard for parallel programming of heterogeneous systems*. http://www.khronos.org/opencl.

[16] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40:195–206, August 2010.

[17] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[18] InfiniBand Trade Association. *InfiniBand Architecture Specification, Volume 1 - General Specification, Release 1.2.1*, 2007.

[19] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Berkeley, CA, USA, 2011. USENIX Association.

[20] F. Ji, H. Lin, and X. Ma. RSVM: a region-based software virtual memory for GPU. In *Proceedings of 22nd International Confreence on Parallel Architectures and Compilation Techniques (PACT)*, pages 269–278. IEEE, 2013.

[21] S. Kato, J. Aumiller, and S. Brandt. Zero-copy I/O processing for low-latency GPU computing. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 170–178, New York, NY, USA, 2013. ACM.

[22] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2011. USENIX Association.

[23] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[24] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2007. USENIX Association.

[25] F. X. Lin, Z. Wang, and L. Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2014.

[26] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM Operating Systems Review*, 43(4):92–105, 2010.

[27] S. Potluri, D. Bureddy, K. Hamidouche, A. Venkatesh, K. Kandalla, H. Subramoni, and D. K. Panda. MVAPICH-PRISM: A proxy-based communication framework using infiniband and SCIF for Intel MIC clusters. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, New York, NY, USA, 2013. ACM.

[28] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 80–89. IEEE, 2013.

[29] A. Rasmussen, M. Conley, R. Kapoor, V. T. Lam, G. Porter, and A. Vahdat. Themis: An I/O Efficient MapReduce. In *Proceedings of the ACM Symposium on Cloud Computing*, 2012.

[30] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 233–248, 2011.

[31] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 49–68, New York, NY, USA, 2013. ACM.

[32] Sean Hefty. Rsockets. OpenFabrics International Workshop, 2012. `https://www.openfabrics.org/index.php/resources/document-downloads/public-documents/doc_download/495-rsockets.html`.

[33] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. Isostack: Highly efficient network processing on dedicated cores. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2010. USENIX Association.

[34] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating file systems with GPUs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013.

[35] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating file systems with GPUs. *ACM Transactions on Computer Systems (TOCS)*, 2014.

[36] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1068–1079. IEEE, 2011.

[37] W. Sun and R. Ricci. Fast and Flexible: Parallel packet processing with GPUs and Click. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 25–36, Piscataway, NJ, USA, 2013. IEEE Press.

[38] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 9–16. ACM, 2011.

[39] A. Trivedi, B. Metzler, P. Stuedi, and T. R. Gross. On limitations of network acceleration. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 121–126, New York, NY, USA, 2013. ACM.

[40] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis. Gaspp: A gpu-accelerated stateful packet processing framework. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 321–332, Philadelphia, PA, June 2014. USENIX Association.

[41] Vasily Volkov. Better performance at lower occupancy. GPU Technology Conference, 2010. `http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf`.

[42] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. In *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA, 2012. ACM.

[43] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *ACM Operating Systems Review*, volume 37, pages 268–281. ACM, 2003.

[44] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

[45] B. Woodruf. OFS software for the Intel Xeon Phi. OpenFabrics Alliance International Developer Workshop, 2013.

The Mystery Machine: End-to-end performance analysis of large-scale Internet services

Michael Chow, David Meisner*†*, Jason Flinn*, Daniel Peek*†*, Thomas F. Wenisch**
University of Michigan   Facebook, Inc.*†*

Abstract

Current debugging and optimization methods scale poorly to deal with the complexity of modern Internet services, in which a single request triggers parallel execution of numerous heterogeneous software components over a distributed set of computers. The Achilles' heel of current methods is the need for a complete and accurate model of the system under observation: producing such a model is challenging because it requires either assimilating the collective knowledge of hundreds of programmers responsible for the individual components or restricting the ways in which components interact.

Fortunately, the scale of modern Internet services offers a compensating benefit: the sheer volume of requests serviced means that, even at low sampling rates, one can gather a tremendous amount of empirical performance observations and apply "big data" techniques to analyze those observations. In this paper, we show how one can automatically construct a model of request execution from pre-existing component logs by generating a large number of potential hypotheses about program behavior and rejecting hypotheses contradicted by the empirical observations. We also show how one can validate potential performance improvements without costly implementation effort by leveraging the variation in component behavior that arises naturally over large numbers of requests to measure the impact of optimizing individual components or changing scheduling behavior.

We validate our methodology by analyzing performance traces of over 1.3 million requests to Facebook servers. We present a detailed study of the factors that affect the end-to-end latency of such requests. We also use our methodology to suggest and validate a scheduling optimization for improving Facebook request latency.

1 Introduction

There is a rich history of systems that understand, optimize, and troubleshoot software performance, both in practice and in the research literature. Yet, most of these prior systems deal poorly with the complexities that arise from modern Internet service infrastructure. Complexity comes partially from *scale*; a single Web request may trigger the execution of hundreds of executable components running in parallel on many different computers. Complexity also arises from *heterogeneity*; executable components are often written in different languages, communicate through a wide variety of channels, and run in execution environments that range from third-party browsers to open-source middleware to in-house, custom platforms.

In this paper, we develop performance analysis tools for measuring and uncovering performance insights about complex, heterogeneous distributed systems. We apply these tools to the Facebook Web pipeline. Specifically, we measure *end-to-end performance* from the point when a user initiates a page load in a client Web browser, through server-side processing, network transmission, and JavaScript execution, to the point when the client Web browser finishes rendering the page.

Fundamentally, analyzing the performance of concurrent systems requires a model of application behavior that includes the causal relationships between components; e.g., *happens-before* ordering and mutual exclusion. While the techniques for performing such analysis (e.g., critical path analysis) are well-understood, prior systems make assumptions about the ease of generating the causal model that simply do not hold in many large-scale, heterogeneous distributed systems such as the one we study in this paper.

Many prior systems assume that one can generate such a model by comprehensively instrumenting all middleware for communication, scheduling, and/or synchronization to record component interactions [1, 3, 13, 18, 22, 24, 28]. This is a reasonable assumption if the software architecture is homogeneous; for instance, Dapper [28] instruments a small set of middleware components that are widely used within Google.

However, many systems are like the Facebook systems we study; they grow organically over time in a culture that favors innovation over standardization (e.g., "move fast and break things" is a well-known Facebook slogan). There is broad diversity in programming languages, communication middleware, execution environments, and scheduling mechanisms. Adding instrumentation retroactively to such an infrastructure is a Herculean task. Further, the end-to-end pipeline includes client software such as Web browsers, and adding detailed instrumentation to all such software is not feasible.

Other prior systems rely on a user-supplied schema that expresses the causal model of application behav-

ior [6, 31]. This approach runs afoul of the scale of modern Internet services. To obtain a detailed model of end-to-end request processing, one must assemble the collective knowledge of hundreds of programmers responsible for the individual components that are involved in request processing. Further, any such model soon grows stale due to the constant evolution of the system under observation, and so constant updating is required.

Consequently, we develop a technique that generates a causal model of system behavior without the need to add substantial new instrumentation or manually generate a schema of application behavior. Instead, we generate the model via large-scale reasoning over individual software component logs. Our key observation is that the sheer volume of requests handled by modern services allows us to gather observations of the order in which messages are logged over a tremendous number of requests. We can then hypothesize and confirm relationships among those messages. We demonstrate the efficacy of this technique with an implementation that analyzes over 1.3 million Facebook requests to generate a comprehensive model of end-to-end request processing.

Logging is an almost-universally deployed tool for analysis of production software. Indeed, although there was no comprehensive tracing infrastructure at Facebook prior to our work, almost all software components had some individual tracing mechanism. By relying on only a minimum common content for component log messages (a request identifier, a host identifier, a host-local timestamp, and a unique event label), we unified the output from diverse component logs into a unified tracing system called *ÜberTrace*.

ÜberTrace's objective is to monitor *end-to-end request latency*, which we define to be the time that elapses from the moment the user initiates a Facebook Web request to the moment when the resulting page finishes rendering. *ÜberTrace* monitors a diverse set of activities that occur on the client, in the network and proxy layers, and on servers in Facebook data centers. These activities exhibit a high degree of concurrency.

To understand concurrent component interactions, we construct a causality model from a large corpus of *ÜberTrace* traces. We generate a cross-product of possible hypotheses for relationships among the individual component events according to standard patterns (currently, happens-before, mutual exclusive, and first-in-first-out relationships). We assume that a relationship holds until we observe an explicit contradiction. Our results show that this process requires traces of hundreds of thousands of requests to converge on a model. However, for a service such as Facebook, it is trivial to gather traces at this scale even at extremely low sampling frequencies. Further, the analysis scales well and runs as a parallel Hadoop job.

Thus, our analysis framework, *The Mystery Machine* derives its causal model solely from empirical observations that utilize only the existing heterogeneous component logs. *The Mystery Machine* uses this model to perform standard analyses, such as identifying critical paths, slack analysis, and outlier detection.

In this paper, we also present a detailed case study of performance optimization based on results from *The Mystery Machine*. First, we note that whereas the average request workload shows a balance between client, server, and network time on the critical path, there is wide variance in this balance across individual requests. In particular, we demonstrate that Facebook servers have considerable slack when processing some requests, but they have almost no slack for other requests. This observation suggests that end-to-end latency would be improved by having servers produce elements of the response as they are needed, rather than trying to produce all elements as fast as possible. We conjecture that this just-in-time approach to response generation will improve the end-to-end latency of requests with no slack while not substantially degrading the latency of requests that currently have considerable slack.

Implementing such an optimization is a formidable task, requiring substantial programming effort. To help justify this cost by partially validating our conjecture, we use *The Mystery Machine* to perform a "what-if" analysis. We use the inherent variation in server processing time that arises naturally over a large number of requests to show that increasing server latency has little effect on end-to-end latency when slack is high. Yet, increasing server latency has an almost linear effect on end-to-end latency when slack is low. Further, we show that slack can be predicted with reasonable accuracy. Thus, the case study demonstrates two separate benefits of *The Mystery Machine*: (1) it can identify opportunities for performance improvement, and (2) it can provide preliminary evidence about the efficacy of hypothesized improvements prior to costly implementation.

2 Background

In the early days of the Web, a request could often be modeled as a single logical thread of control in which a client executed an RPC to a single Web server. Those halcyon days are over.

At Facebook, the end-to-end path from button click to final render spans a diverse set of systems. Many components of the request are under Facebook's control, but several components are not (e.g., the external network and the client's Web browser). Yet, users care little about who is responsible for each component; they simply desire that their content loads with acceptable delay.

A request begins on a client with a user action to retrieve some piece of content (e.g., a news feed). After

DNS resolution, the request is routed to an Edge Load Balancer (ELB) [16]. ELBs are geo-distributed so as to allow TCP sessions to be established closer to the user and avoid excessive latency during TCP handshake and SSL termination. ELBs also provide a point of indirection for better load balancing, acting as a proxy between the user and data center.

Once a request is routed to a particular data center, a Software Load Balancer routes it to one of many possible Web servers, each of which runs the HipHop Virtual Machine runtime [35]. Request execution on the Web server triggers many RPCs to caching layers that include Memcache [20] and TAO [7]. Requests also occasionally access databases.

RPC responses pass through the load-balancing layers on their way back to the client. On the client, the exact order and manner of rendering a Web page are dependent on the implementation details of the user's browser. However, in general, there will be a Cascading Style Sheet (CSS) download stage and a Document Object Model rendering stage, followed by a JavaScript execution stage.

As with all modern Internet services, to achieve latency objectives, the handling of an individual request exhibits a high degree of concurrency. Tens to hundreds of individual components execute in parallel over a distributed set of computers, including both server and client machines. Such concurrency makes performance analysis and debugging complex. Fortunately, standard techniques such as critical path analysis and slack analysis can tame this complexity. However, all such analyses need a model of the causal dependencies in the system being analyzed. Our work fills this need.

3 *ÜberTrace*: End-to-end Request Tracing

As discussed in the prior section, request execution at Facebook involves many software components. Prior to our work, almost all of these components had logging mechanisms used for debugging and optimizing the individual components. In fact, our results show that individual components are almost always well-optimized *when considered in isolation*.

Yet, there existed no complete and detailed instrumentation for monitoring the end-to-end performance of Facebook requests. Such end-to-end monitoring is vital because individual components can be well-optimized in isolation yet still miss opportunities to improve performance when components interact. Indeed, the opportunities for performance improvement we identify all involve the interaction of multiple components.

Thus, the first step in our work was to unify the individual logging systems at Facebook into a single end-to-end performance tracing tool, dubbed *ÜberTrace*. Our basic approach is to define a minimal schema for the information contained in a log message, and then map existing log messages to that schema.

ÜberTrace requires that log messages contain at least:

1. A unique request identifier.
2. The executing computer (e.g., the client or a particular server)
3. A timestamp that uses the local clock of the executing computer
4. An event name (e.g., "start of DOM rendering").
5. A task name, where a task is defined to be a distributed thread of control.

ÜberTrace requires that each <event, task> tuple is unique, which implies that there are no cycles that would cause a tuple to appear multiple times. Although this assumption is not valid for all execution environments, it holds at Facebook given how requests are processed. We believe that it is also a reasonable assumption for similar Internet service pipelines.

Since all log timestamps are in relation to local clocks, *ÜberTrace* translates them to estimated global clock values by compensating for clock skew. *ÜberTrace* looks for the common RPC pattern of communication in which the thread of control in an individual task passes from one computer (called the client to simplify this explanation) to another, executes on the second computer (called the server), and returns to the client. *ÜberTrace* calculates the server execution time by subtracting the latest and earliest server timestamps (according to the server's local clock) nested within the client RPC. It then calculates the client-observed execution time by subtracting the client timestamps that immediately succeed and precede the RPC. The difference between the client and server intervals is the estimated network round-trip time (RTT) between the client and server. By assuming that request and response delays are symmetric, *ÜberTrace* calculates clock skew such that, after clock-skew adjustment, the first server timestamp in the pattern is exactly 1/2 RTT after the previous client timestamp for the task.

The above methodology is subject to normal variation in network performance. In addition, the imprecision of using existing log messages rather than instrumenting communication points can add uncertainty. For instance, the first logged server message could occur only after substantial server execution has already completed, leading to an under-estimation of server processing time and an over-estimation of RTT. *ÜberTrace* compensates by calculating multiple estimates. Since there are many request and response messages during the processing of a higher-level request, it makes separate RTT and clock

skew calculations for each pair in the cross-product of requests. It then uses the calculation that yields the lowest observed RTT.

Timecard [23] used a similar approach to reconcile timestamps and identified the need to account for the effects of TCP slow start. Our use of multiple RTT estimates accomplishes this. Some messages such as the initial request are a single packet and so are not affected by slow start. Other messages such as the later responses occur after slow start has terminated. Pairing two such messages will therefore yield a lower RTT estimate. Since we take the minimum of the observed RTTs and use its corresponding skew estimate, we get an estimate that is not perturbed by slow start.

Due to performance considerations, Facebook logging systems use statistical sampling to monitor only a small percentage of requests. *ÜberTrace* must ensure that the individual logging systems choose the same set of requests to monitor; otherwise the probability of all logging systems independently choosing to monitor the same request would be vanishingly small, making it infeasible to build a detailed picture of end-to-end latency. Therefore, *ÜberTrace* propagates the decision about whether or not to monitor a request from the initial logging component that makes such a decision through all logging systems along the path of the request, ensuring that the request is completely logged. The decision to log a request is made when the request is received at the Facebook Web server; the decision is included as part of the per-request metadata that is read by all subsequent components. *ÜberTrace* uses a global identifier to collect the individual log messages, extracts the data items enumerated above, and stores each message as a record in a relational database.

We made minimal changes to existing logging systems in order to map existing log messages to the *ÜberTrace* schema. We modified log messages to use the same global identifier, and we made the event or task name more human-readable. We added no additional log messages. Because we reused existing component logging and required only a minimal schema, these logging changes required approximately one person-month of effort.

4 *The Mystery Machine*

The Mystery Machine uses the traces generated by *ÜberTrace* to create a causal model of how software components interact during the end-to-end processing of a Facebook request. It then uses the causal model to perform several types of distributed systems performance analysis: finding the critical path, quantifying slack for segments not on the critical path, and identifying segments that are correlated with performance anomalies. *The Mystery Machine* enables more targeted analysis by exporting its results through a relational database and graphical query tools.

Relationship	Example	Counterexample
Happens Before	A B, t →	A, B or B A
Mutual Exclusion	A B or B A	A, B
Pipeline	t_1 A B C, t_2 A' B' C'	t_1 A B C, t_2 C' A' B'

Figure 1: Causal Relationships. This figure depicts examples of the three kinds of causal relationship we consider. Happens-before relationships are when one segment (A) always finishes in its entirety before another segment (B) begins. FIFO relationships exist when a sequence of segments each have a happens-before relationship with another sequence in the same order. A mutual exclusion relationship exists when two segments never overlap.

4.1 Causal Relationships Model

To generate a causal model, *The Mystery Machine* first transforms each trace from a collection of logged events to a collection of *segments*, which we define to be the execution interval between two consecutive logged events for the same task. A segment is labeled by the tuple <task, start_event, end_event>, and the segment duration is the time interval between the two events.

Next, *The Mystery Machine* identifies causal relationships. Currently, it looks for three types of relationships:

1. **Happens-before** (→) We say that segment A happens-before segment B (A → B) if the start event timestamp for B is greater than or equal to the end event timestamp for A in all requests.

2. **Mutual exclusion** (∨) Segments A and B are mutually exclusive (A ∨ B) if their time intervals never overlap.

3. **Pipeline** (≫) Given two tasks, t_1 and t_2, there exists a data dependency between pairs of segments of the two tasks. Further, the segment that operates on data element d_1 precedes the segment that operates on data element d_2 in task t_1 if and only if the segment that operates on d_1 precedes the segment that operates on d_2 in task t_2 for all such pairs of segments. In other words, the segments preserve a FIFO ordering in how data is produced by the first task and consumed by the second task.

We summarize these relationships in Figure 1. For each relationship we provide a valid example and at least one counterexample that would contradict the hypothesis.

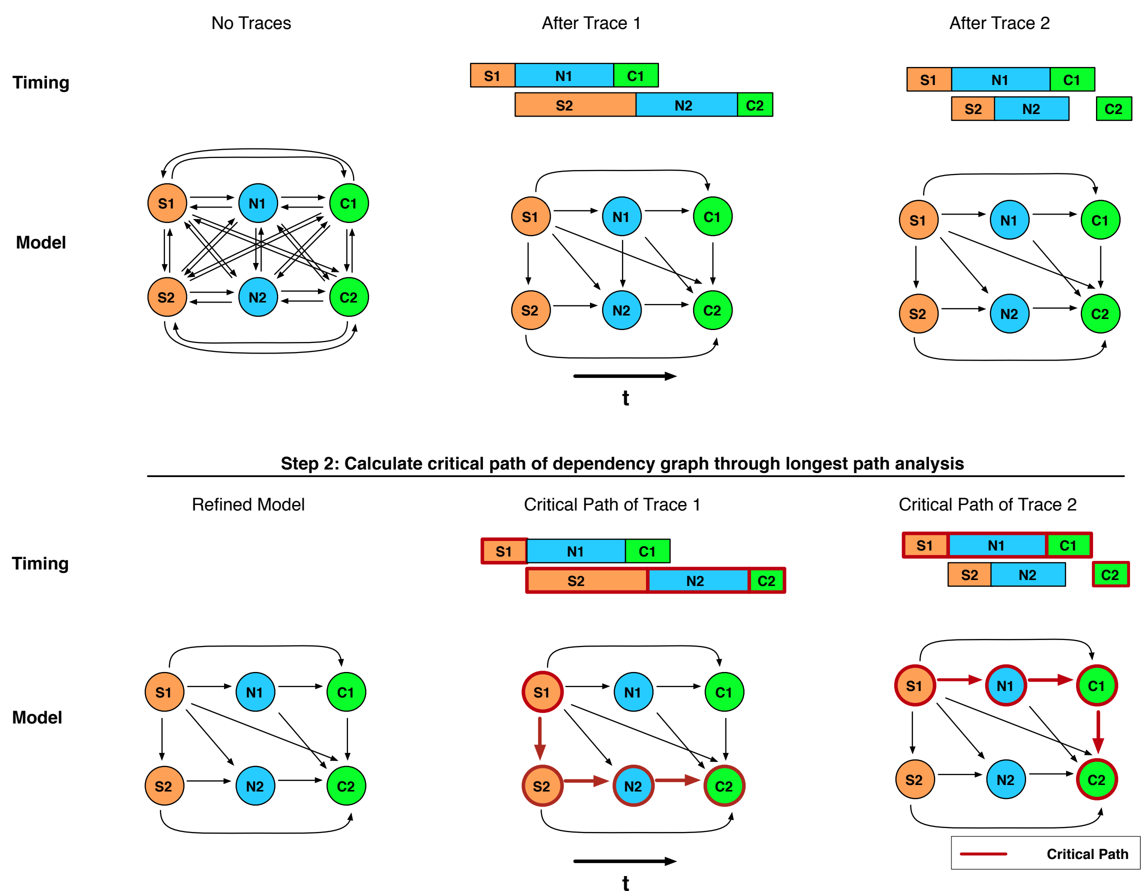


Figure 2: Dependency model generation and critical path calculation. This figure provides an example of discovering the true dependency model through iterative refinement. We show only a few segments and relationships for the sake of simplicity. Without any traces, the dependency model is a fully connected graph. By eliminating dependency edges invalidated by counterexamples, we arrive at the true model. With a refined model, we can reprocess the same traces and derive the critical path for each.

We use techniques from the race detection literature to map these static relationships to dynamic happens-before relationships. Note that mutual exclusion is a static property; e.g., two components A and B that share a lock are mutually exclusive. Dynamically, for a particular request, this relationship becomes a happens-before relationship: either A → B or B → A, depending on the order of execution. Pipeline relationships are similar. Thus, for any given request, all of these static relationships can be expressed as dynamic causal relationships between pairs of segments.

4.2 Algorithm

The Mystery Machine uses iterative refinement to infer causal relationships. It first generates all possible hypotheses for causal relationships among segments. Then, it iterates through a corpus of traces and rejects a hypothesis if it finds a counterexample in any trace.

Step 1 of Figure 2 illustrates this process. We depict the set of hypotheses as a graph where nodes are segments ("S" nodes are server segments, "N" nodes are network segments and "C" nodes are client segments) and edges are hypothesized relationships. For the sake of simplicity, we restrict this example to consider only happens-before relationships; an arrow from A to B shows a hypothesized "A happens before B" relationship.

The "No Traces" column shows that all possible relationships are initially hypothesized; this is a large number because the possible relationships scale quadratically as the number of segments increases. Several hypotheses are eliminated by observed contradictions in the first request. For example, since S2 happens after S1, the hypothesized relationship, S2 → S1, is removed. Further traces must be processed to complete the model. For instance, the second request eliminates the hypothesized relationship, N1 → N2. Additional traces prune new hypotheses due to the natural perturbation in timing of segment processing; e.g., perhaps the second user had less friends, allowing the network segments to overlap due to

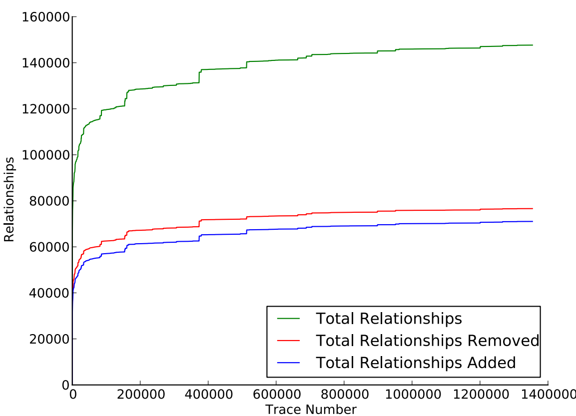


Figure 3: Hypothesis Refinement. This graph shows the growth of number of hypothesized relationships as a function of requests analyzed. As more requests are analyzed, the rate at which new relationships are discovered and removed decreases and eventually reaches a steady-state. The total number of relationships increases over time due to code changes and the addition of new features.

shorter server processing time.

The Mystery Machine assumes that the natural variation in timing that arises over large numbers of traces is sufficient to expose counterexamples for incorrect relationships. Figure 3 provides evidence supporting this hypothesis from traces of over 1.3 million requests to the Facebook home page gathered over 30 days. As the number of traces analyzed increases, the observation of new counterexamples diminishes, leaving behind only true relationships. Note that the number of total relationships changes over time because developers are continually adding new segments to the pipeline.

4.3 Validation

To validate the causal model produced by the Mystery Machine, we confirmed several specific relationships identified by the Mystery Machine. Although we could not validate the entire model due to its size, we did substantial validation of two of the more intricate components: the interplay between JavaScript execution on the client and the dependencies involved in delivering data to the client. These components have 42 and 84 segments, respectively, as well as 2,583 and 10,458 identified casual relationships.

We confirmed these specific relationships by examining source code, inserting assertions to confirm model-derived hypotheses, and consulting relevant subsystem experts. For example, the system discovered the specific, pipelined schedule according to which page content is delivered to the client. Further, the model correctly reflects that JavaScript segments are mutually exclusive (a known property of the JavaScript execution engine) and identified ordering constraints arising from synchronization.

4.4 Analysis

Once *The Mystery Machine* has produced the causal model of segment relationships, it can perform several types of performance analysis.

4.4.1 Critical Path

Critical path analysis is a classic technique for understanding how individual components of a parallel execution impact end-to-end latency [22, 32]. The critical path is defined to be the set of segments for which a differential increase in segment execution time would result in the same differential increase in end-to-end latency.

The Mystery Machine calculates the critical path on a per-request basis. It represents all segments in a request as a directed acyclic graph in which the segments are vertices with weight equal to the segment duration. It adds an edge between all vertices for which the corresponding segments have a causal relationship. Then, it performs a transitive reduction in which all edges A $\rightarrow$ C are recursively removed if there exists a path consisting of A $\rightarrow$ B and B $\rightarrow$ C that links the two nodes.

Finally, *The Mystery Machine* performs a longest-path analysis to find the critical path from the first event in the request (the initiation of the request) to the last event (which is typically the termination of some JavaScript execution). The length of the critical path is the end-to-end latency of the entire request. If there are equal-length critical paths, the first discovered path is chosen.

We illustrate the critical path calculation for the two example requests in Step 2 of Figure 2. Each request has a different critical path even though the dependency graph is the same for both. The critical path of the first request is $\{$S1, S2, N2, C2$\}$. Because S2 has a long duration, all dependencies for N2 and C2 have been met before they start, leaving them on the critical path. The critical path of the second request is $\{$S1, N1, C1, C2$\}$. In this case, S2 and N2 could have longer durations and not affect end-to-end latency because C2 must wait for C1 to finish.

Typically, we ask *The Mystery Machine* to calculate critical paths for large numbers of traces and aggregate the results. For instance, we might ask how often a given segment falls on the critical path or the average percentage of the critical path represented by each segment.

4.4.2 Slack

Critical path analysis is useful for determining where to focus optimization effort; however, it does not provide any information about the importance of latency for segments off the critical path. *The Mystery Machine* provides this information via slack analysis.

We define slack to be the amount by which the duration of a segment may increase without increasing the

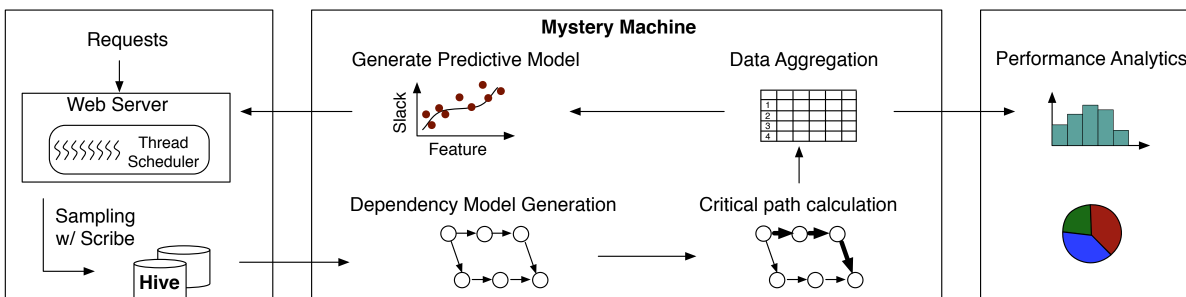


Figure 4: *The Mystery Machine* data pipeline.

end-to-end latency of the request, assuming that the duration of all other segments remains constant. By this definition, segments on the critical path have no slack because increasing their latency will increase the end-to-end latency of the request.

To calculate the slack for a given segment, S, *The Mystery Machine* calculates CP_{start}, the critical path length from the first event in the request to the start of S and CP_{end} the critical path length from the end of S to the last event in the request. Given the critical path length for the entire request (CP) and the duration of segment S (D_S), the slack for S is CP - CP_{start} - D_S - CP_{end}. *The Mystery Machine*'s slack analysis calculates and reports this value for every segment. As with critical path results, slack results are typically aggregated over a large number of traces.

4.4.3 Anomaly detection

One special form of aggregation supported by *The Mystery Machine* is anomaly analysis. To perform this analysis, it first classifies requests according to end-to-end latency to identify a set of outlier requests. Currently, outliers are defined to be requests that are in the top 5% of end-to-end latency. Then, it performs a separate aggregation of critical path or slack data for each set of requests identified by the classifiers. Finally, it performs a differential comparison to identify segments with proportionally greater representation in the outlier set of requests than in the non-outlier set. For instance, we have used this analysis to identify a set of segments that correlated with high latency requests. Inspection revealed that these segments were in fact debugging components that had been returned in response to some user requests.

4.5 Implementation

We designed *The Mystery Machine* to automatically and continuously analyze production traffic at scale over long time periods. It is implemented as a large-scale data processing pipeline, as depicted in Figure 4.

ÜberTrace continuously samples a small fraction of requests for end-to-end tracing. Trace data is collected by the Web servers handling these requests, which write them to Scribe, Facebook's distributed logging service. The trace logs are stored in tables in a large-scale data warehousing infrastructure called Hive [30]. While Scribe and Hive are the in-house analysis tools used at Facebook, their use is not fundamental to our system.

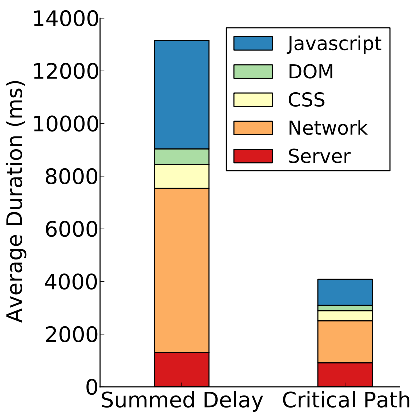
The Mystery Machine runs periodic processing jobs that read trace data from Hive and calculate or refine the causal model based on those traces. The calculation of the causal model is compute-intensive because the number of possible hypotheses is quadratic with the number of segments and because model refinement requires traces of hundreds of thousands of requests. Therefore, our implementation parallelizes this step as a Hadoop job running on a compute cluster. Infrequently occurring testing and debugging segments are automatically removed from the model; these follow a well-defined naming convention that can be detected with a single regular expression. The initial calculation of the model analyzed traces of over 1.3 million requests collected over 30 days. On a Hadoop cluster, it took less than 2 hours to derive a model from these traces.

In practice, the model must be recomputed periodically in order to detect changes in relationships. Parallelizing the computation made it feasible to recompute the model every night as a regularly-scheduled batch job.

In addition to the three types of analysis described above, *The Mystery Machine* supports on-demand user queries by exporting results to Facebook's in-house analytic tools, which can aggregate, pivot, and drill down into the results. We used these tools to categorize results by browser, connection speed, and other such dimensions; we share some of this data in Section 5.

4.6 Discussion

A key characteristic of *The Mystery Machine* is that it discovers dependencies automatically, which is critical because Facebook's request processing is constantly evolving. As described previously, *The Mystery Machine* assumes a hypothesized relationship between two segments until it finds a counterexample. Over time, new segments are added as the site evolves and new features are added. *The Mystery Machine* automatically finds the dependencies introduced by the new segments by hy-



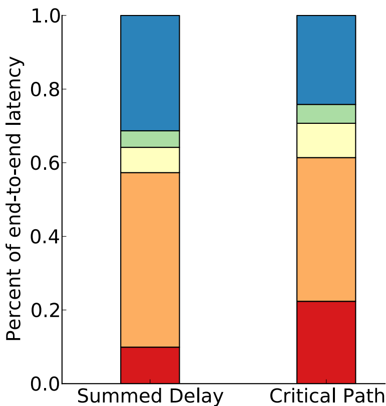


Figure 5: Mean End-to-End Performance Breakdown. Simply summing delay measured at each system component ("Summed Delay") ignores overlap and underestimates the importance of server latency relative to the actual mean critical path ("Critical Path").

pothesizing new possible relationships and removing relationships in which a counterexample is found. This is shown in Figure 3 by the increase in number of total relationships over time. To account for segments that are eliminated and invariants that are added, one can simply run a new Hadoop job to generate the model over a different time window of traces.

Excluding new segments, the rate at which new relationships are added levels off. The rate at which relationships are removed due to counterexamples also levels off. Thus, the model converges on a set of true dependencies.

The Mystery Machine relies on *ÜberTrace* for complete log messages. Log messages, however, may be missing for two reasons: the component does no logging at all for a segment of its execution or the component logs messages for some requests but not others. In the first case, *The Mystery Machine* cannot identify causal relationships involving the unlogged segment, but causal relationships among all other segments will be identified correctly. When a segment is missing, the model overestimates the concurrency in the system, which would affect the critical path/slack analysis if the true critical path includes the unlogged segment. In the second case, *The Mystery Machine* would require more traces in order to discover counterexamples. This is equivalent to changing the sampling frequency.

5 Results

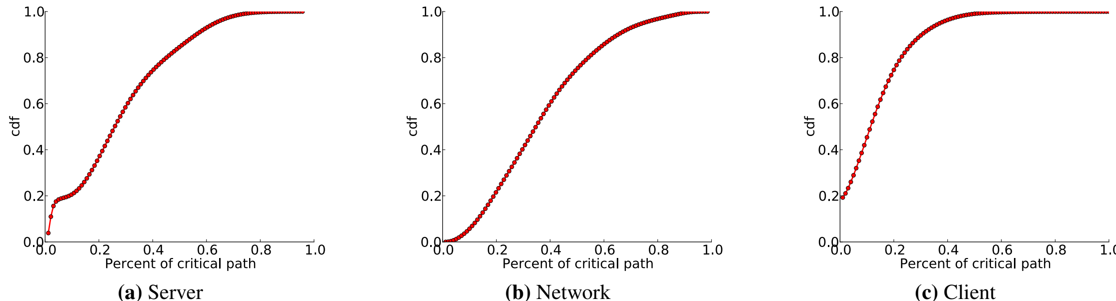
We demonstrate the utility of *The Mystery Machine* with two case studies. First, we demonstrate its use for aggregate performance characterization. We study live traffic, stratifying the data to identify factors that influence which system components contribute to the critical path. We find that the critical path can shift between three major components (servers, network, and client) and that these shifts correlate with the client type and network connection quality.

This variation suggests one possible performance optimization for Facebook servers: provide differentiated service by prioritizing service for connections where the server has no slack while deprioritizing those where network and client latency will likely dominate. Our second case study demonstrates how the natural variance across a large trace set enables testing of such performance hypotheses without expensive modifications to the system under observation. Since an implementation that provided differential services would require large-scale effort to thread through hundreds of server components, we use our dataset to first determine whether such an optimization is likely to be successful. We find that slack, as detected by *The Mystery Machine*, indeed indicates that slower server processing time minimally impacts end-to-end latency. We also find that slack tends to remain stable for a particular user across multiple Facebook sessions, so the observed slack of past connections can be used to predict the slack of the current connection.

5.1 Characterizing End-to-End Performance

In our first case study, we characterize the end-to-end performance critical path of Web accesses to the `home.php` Facebook endpoint. *The Mystery Machine* analyzes traces of over 1.3 million Web accesses collected over 30 days in July and August 2013.

Importance of critical path analysis. Figure 5 shows mean time breakdowns over the entire trace dataset. The breakdown is shown in absolute time in the left graph, and as a percent of total time on the right. We assign segments to one of five categories: *Server* for segments on a Facebook Web server or any internal service accessed from the Web server over RPC, *Network* for segments in which data traverses the network, *DOM* for



(a) Server **(b)** Network **(c)** Client

Figure 6: Cumulative distribution of the fraction of the critical path attributable to server, network, and client portions

browser segments that parse the document object model, *CSS* for segments processing cascading style sheets, and *JavaScript* for JavaScript segments. Each graph includes two bars: one showing the stacked sum of total processing time in each component ignoring all concurrency ("Summed Delay") and the other the critical path as identified by *The Mystery Machine* ("Critical Path").

On average, network delays account for the largest fraction of the critical path, but client and server processing are both significant. JavaScript execution remains a major bottleneck in current Web browsers, particularly since the JavaScript execution model admits little concurrency. The comparison of the total delay and critical path bars reveals the importance of *The Mystery Machine*—by examining only the total latency breakdown (e.g., if an engineer were profiling only one system component), one might overestimate the importance of network latency and JavaScript processing on end-to-end performance. In fact, the server and other client processing segments are frequently critical, and the overall critical path is relatively balanced across server, client, and network.

High variance in the critical path. Although analyzing the average case is instructive, it grossly oversimplifies the performance picture for the `home.php` endpoint. There are massive sources of latency variance over the population of requests, including the performance of the client device, the size of the user's friend list, the kind of network connection, server load, Memcache misses, etc. Figure 6 shows the cumulative distribution of the fraction of the critical path attributable to server, network, and client segments over all requests. The key revelation of these distributions is that the critical path shifts drastically across requests—any of the three components can dominate delay, accounting for more than half of the critical path in a non-negligible fraction of requests.

Variance is greatest in the contribution of the network to the critical path, as evidenced by the fact that its CDF has the least curvature. It is not surprising that network delays vary so greatly since the trace data set includes accesses to Facebook over all sorts of networks, from high-speed broadband to cellular networks and even some dial-up connections. Client processing always accounts for at least 20% of the critical path. After content delivery, there is a global barrier in the browser before the JavaScript engine begins running the executable components of the page, hence, JavaScript execution is a factor in performance measurement. However, the client rarely accounts for more than 40% of the critical path. It is unusual for the server to account for less than 20% of the critical path because the initial request processing before the server begins to transmit any data is always critical. Noticing this high variance in the critical path was very valuable to us because it triggered the idea of differentiated services that we explore in Section 5.2.

Stratification by connection type. We first consider stratifying by the type of network over which a user connects to Facebook's system, as it is clear one would expect network latency to differ, for example, between cable modem and wireless connections. Facebook's edge load balancing system tags each incoming request with a network type. These tags are derived from the network type recorded in the Autonomous System Number database for the Internet service provider responsible for the originating IP address. Figure 7 illustrates the critical path breakdown, in absolute time, for the four largest connection type categories. Each bar is annotated with the fraction of all requests that fall within that connection type (only a subset of connection types are shown, so the percentages do not sum to 100%).

Perhaps unsurprisingly, these coarse network type classifications correlate only loosely to the actual performance of the network connection. Mobile connections show a higher average network critical path than the other displayed connection types, but the data is otherwise inconclusive. We conclude that the network type reported by the ASN is not very helpful for making performance predictions.

Stratification by client platform. The client platform is included in the HTTP headers transmitted by the browser along with each request, and is therefore also available at the beginning of request processing. The

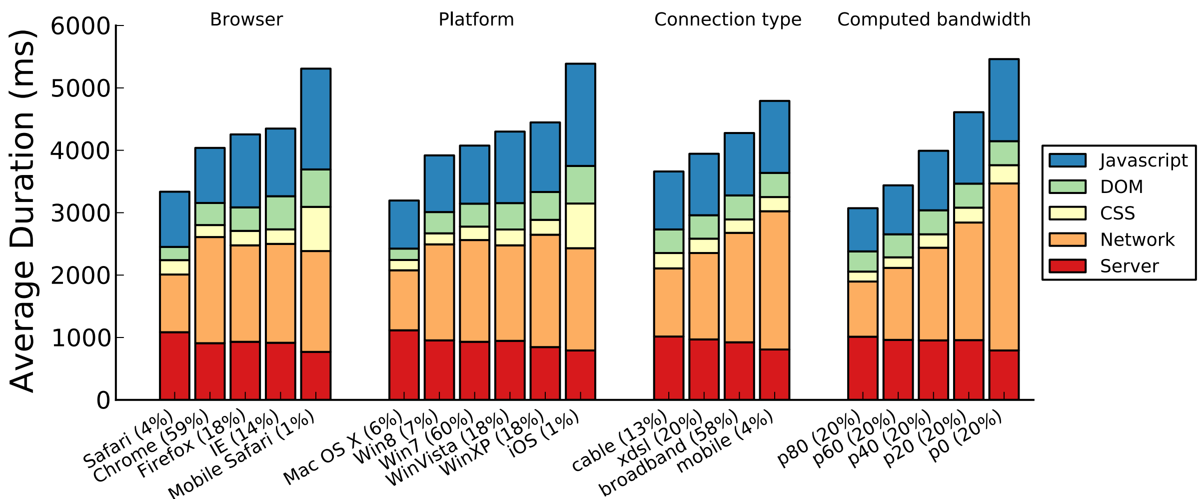


Figure 7: Critical path breakdowns stratified by browser, platform, connection type, and computed bandwidth

client operating system is a hint to the kind of client device, which in turn may suggest relative client performance. Figure 7 shows a critical path breakdown for the five most common client platforms in our traces, again annotated with the fraction of requests represented by the bar. Note that we are considering only Web browser requests, so requests initiated by Facebook cell phone apps are not included. The most striking feature of the graph is that Mac OS X users (a small minority of Facebook connections at only 7.1%) tend to connect to Facebook from faster networks than Windows users. We also see that the bulk of connecting Windows users still run Windows 7, and many installations of Windows XP remain deployed. Client processing time has improved markedly over the various generations of Windows. Nevertheless, the breakdowns are all quite similar, and we again find insufficient predictive power for differentiating service time by platform.

Stratification by browser. The browser type is also indicated in the HTTP headers transmitted with a request. In Figure 7, we see critical paths for the four most popular browsers. Safari is an outlier, but this category is strongly correlated with the Mac OS X category. Chrome appears to offer slightly better JavaScript performance than the other browsers.

Stratification by measured network bandwidth. All of the preceding stratifications only loosely correlate to performance—ASN is a poor indication of network connection quality, and browser and OS do not provide a reliable indication of client performance. We provide one more example stratification where we subdivide the population of requests into five categories directly from the measured network bandwidth, which can be deduced from our traces based on network time and

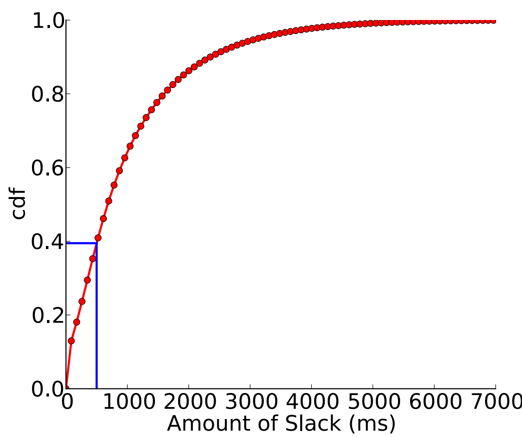


Figure 8: Slack CDF for Last Data Item. Nearly 20% of traces exhibit considerable slack—over 2 s—for the server segment that generates the last pagelet transmitted to the client. Conversely, nearly 20% of traces exhibit little (< 250 ms) slack.

bytes transmitted. Each of the categories are equally sized to represent 20% of requests, sorted by increasing bandwidth (p80 is the quintile with the highest observed bandwidth). As one would expect, network critical path is strongly correlated to measured network bandwidth. Higher bandwidth connections also tend to come from more capable clients; low-performance clients (e.g., smart phones) often connect over poor networks (3G and Edge networks).

5.2 Differentiated Service using Slack

Our second case study uses *The Mystery Machine* to perform early exploration of a potential performance optimization—differentiated service—without undertaking the expense of implementing the optimization.

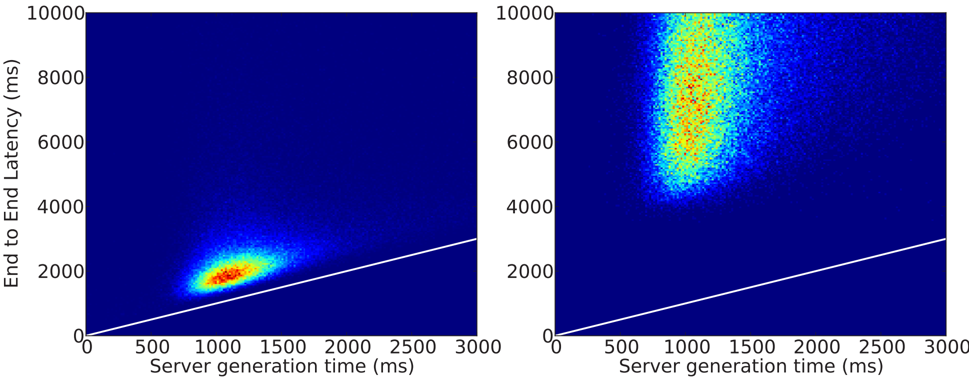


Figure 9: Server vs. End-to-end Latency. For the traces with slack below 25ms (left graph), there is strong correlation (clustering near $y = x$) between server and end-to-end latency. The correlation is much weaker (wide dispersal above $y = x$) for the traces with slack above 2.5s (right graph).

The characterization in the preceding section reveals that there is enormous variation in the relative importance of client, server, and network performance over the population of Facebook requests. For some requests, server segments form the bulk of the critical path. For these requests, any increase in server latency will result in a commensurate increase in end-to-end latency and a worse user experience. However, after the initial critical segment, many connections are limited by the speed at which data can be delivered over the network or rendered by the client. For these connections, server execution can be delayed to produce data as needed, rather than as soon as possible, without affecting the critical path or the end-to-end request latency.

We use *The Mystery Machine* to directly measure the slack in server processing time available in our trace dataset. For simplicity of explanation, we will use the generic term "slack" in this section to refer to the slack in server processing time only, excluding slack available in any other types of segments.

Figure 8 shows the cumulative distribution of slack for the last data item sent by the server to the client. The graph is annotated with a vertical line at 500 ms of slack. For the purposes of this analysis, we have selected 500 ms as a reasonable cut-off between connections for which service should be provided with best effort ($<$ 500 ms slack), and connections for which service can be deprioritized ($>$ 500 ms). However, in practice, the best cut-off will depend on the implementation mechanism used to deprioritize service. More than 60% of all connections exhibit more than 500 ms of slack, indicating substantial opportunity to defer server processing. We find that slack typically increases monotonically during server processing as data items are sent to the client during a request. Thus, we conclude that slack is best consumed equally as several segments execute, as opposed to consuming all slack at the start or end of processing.

Validating Slack Estimates It is difficult to directly validate *The Mystery Machine*'s slack estimates, as we can only compute slack once a request has been fully processed. Hence, we cannot retrospectively delay server segments to consume the slack and confirm that the end-to-end latency is unchanged. Such an experiment is difficult even under highly controlled circumstances, since it would require precisely reproducing the conditions of a request over and over while selectively delaying only a few server segments.

Instead, we turn again to the vastness of our trace data set and the natural variance therein to confirm that slack estimates hold predictive power. Intuitively, small slack implies that server latency is strongly correlated to end-to-end latency; indeed, with a slack of zero we expect any increase in server latency to delay end-to-end latency by the same amount. Conversely, when slack is large, we expect little correlation between server latency and end-to-end latency; increases in server latency are largely hidden by other concurrent delays. We validate our notion of slack by directly measuring the correlation of server and end-to-end latency.

Figure 9 provides an intuitive view of the relationship for which we are testing. Each graph is a heat map of server generation time vs. end-to-end latency. The left graph includes only requests with the lowest measured slack, below 25 ms. There are slightly over 115,000 such requests in this data set. For these requests, we expect a strong correlation between server time and end-to-end time. We find that this subset of requests is tightly clustered just above the $y = x$ (indicated by the line in the figure), indicating a strong correlation. The right figure includes roughly 100,000 requests with the greatest slack (above 2500 ms). For these, we expect no particular relationship between server time and end-to-end time (except that end-to-end time must be at least as large as slack, since this is an invariant of request processing).

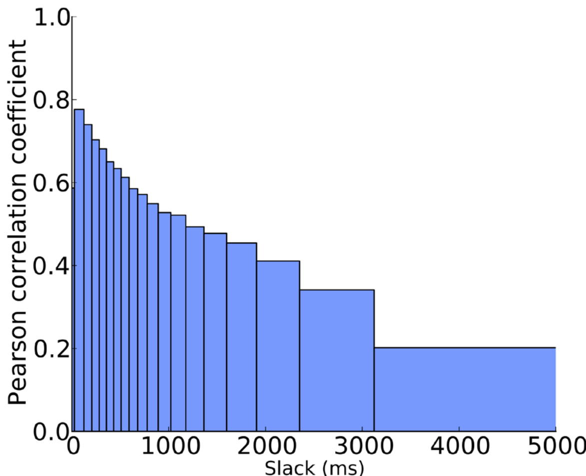


Figure 10: Server–End-to-end Latency Correlation vs. Slack. As reported slack increases, the correlation between total server processing time and end-to-end latency weakens, since a growing fraction of server segments are non-critical.

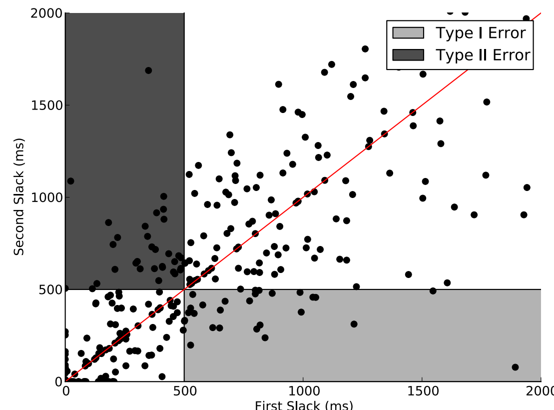


Figure 11: Historical Slack as Classifier. The clustering around the line $y = x$ shows that slack is relatively stable over time. The history-based classifier is correct 83% of the time. A type I error is a false positive, reporting slack as available when it is not. A type II error is a false negative.

Indeed, we find the requests dispersed in a large cloud above $y = x$, with no correlation visually apparent.

We provide a more rigorous validation of the slack estimate in Figure 10. Here, we show the correlation coefficient between server time and end-to-end time for equally sized buckets of requests sorted by increasing slack. Each block in the graph corresponds to 5% of our sample, or roughly 57,000 requests (buckets are not equally spaced since the slack distribution is heavy-tailed). As expected, the correlation coefficient between server and end-to-end latency is quite high, nearly 0.8, when slack is low. It drops to 0.2 for the requests with the largest slack.

Predicting Slack. We have found that slack is predictive of the degree to which server latency impacts end-to-end latency. However, *The Mystery Machine* can discover slack only through a retrospective analysis. To be useful in a deployed system, we must predict the availability or lack of slack for a particular connection as server processing begins.

One mechanism to predict slack is to recall the slack a particular user experienced in a prior connection to Facebook. Previous slack was found to be more useful in predicting future slack than any other feature we studied. Most users connect to Facebook using the same device and over the same network connection repeatedly. Hence, their client and network performance are likely to remain stable over time. The user id is included as part of the request, and slack could be easily associated with the user id via a persistent cookie or by storing the most recent slack estimate in Memcache [20].

We test the hypothesis that slack remains stable over time by finding all instances within our trace dataset where we have multiple requests associated with the same user id. Since the request sampling rate is exceedingly low, and the active user population is so large, selecting the same user for tracing more than once is a relatively rare event. Nevertheless, again because of the massive volume of traces collected over the course of 30 days of sampling, we have traced more than 1000 repeat users. We test a simple classifier that predicts a user will experience a slack greater than 500 ms if the slack on their most recent preceding connection was also greater than 500 ms. Figure 11 illustrates the result. The graph shows a scatter plot of the first slack and second slack in each pair; the line at $y = x$ indicates slack was identical between the two connections. Our simple history-based classifier predicts the presence or absence of slack correctly 83% of the time. The shaded regions of the graph indicate cases where we have misclassified a connection. A type I error indicates a prediction that there is slack available for a connection when in fact server performance turns out to be critical–8% of requests fall in this category. Conversely, a type II error indicates a prediction that a connection will not have slack when in fact it does, and represents a missed opportunity to throttle service—9% of requests fall in this category.

Note that achieving these results does not require frequent sampling. The repeated accesses we study are often several weeks separated in time, and, of course, it is likely that there have been many intervening unsampled requests by the same user. Sampling each user once every few weeks would therefore be sufficient.

Potential Impact. We have shown that a potential performance optimization would be to offer differentiated service based on the predicted amount of slack available per connection. Deciding which connections to service is equivalent to real-time scheduling with deadlines.

By using predicted slack as a scheduling deadline, we can improve average response time in a manner similar to the earliest deadline first real-time scheduling algorithm. Connections with considerable slack can be given a lower priority without affecting end-to-end latency. However, connections with little slack should see an improvement in end-to-end latency because they are given scheduling priority. Therefore, average latency should improve. We have also shown that prior slack values are a good predictor of future slack. When new connections are received, historical values can be retrieved and used in scheduling decisions. Since calculating slack is much less complex than servicing the actual Facebook request, it should be feasible to recalculate the slack for each user approximately once per month.

6 Related Work

Critical path analysis is an intuitive technique for understanding the performance of systems with concurrent activity. It has been applied in a wide variety of areas such as processor design [26], distributed systems [5], and Internet and mobile applications [22, 32].

Deriving the critical path requires knowing causal dependencies between components throughout the entire end-to-end system. A model of causal dependencies can be derived from comprehensively instrumenting all middleware for communication, scheduling, and/or synchronization to record component interactions [1, 3, 9, 13, 15, 18, 22, 24, 28]. In contrast to these prior systems, *The Mystery Machine* is targeted at environments where adding comprehensive new instrumentation to an existing system would be too time-consuming due to heterogeneity (e.g., at Facebook, there a great number of scheduling, communication, and synchronization schemes used during end-to-end request processing) and deployment feasibility (e.g., it is not feasible to add new instrumentation to client machines or third-party Web browser code). Instead, *The Mystery Machine* extracts a causal model from already-existing log messages, relying only a minimal schema for such messages.

Sherlock [4] also uses a "big data" approach to build a causal model. However, it relies on detailed packet traces, not log messages. Packet traces would not serve our purpose: it is infeasible to collect them on user clients, and they reveal nothing about the interaction of software components that run on the same computer (e.g., JavaScript), which is a major focus of our work. Observing a packet sent between A and B inherently implies some causal relationship, while *The Mystery Machine* must infer such relationships by observing if the order of log messages from A and B obey a hypothesized invariant. Hence, Sherlock's algorithm is fundamentally different: it reasons based on temporal locality and infers probabilistic relationships; in contrast, *The Mystery Machine* uses only message order to derive invariants (though timings are used for critical path and slack analysis).

The lprof tool [36] also analyzes log messages to reconstruct the ordering of logged events in a request. It supplements logs with static analysis to discover dependencies between log points and uses those dependencies to differentiate events among requests. Since static analysis is difficult to scale to heterogeneous production environments, *The Mystery Machine* used some manual modifications to map events to traces and leverages a large sample size and natural variation in ordering to infer causal dependencies between events in a request.

In other domains, hypothesizing likely invariants and eliminating those contradicted by observations has proven to be a successful technique. For instance, likely invariants have been used for fault localization [25] and diagnosing software errors [12, 21]. *The Mystery Machine* applies this technique to a new domain.

Many other systems have looked at the notion of critical path in Web services. WebProphet [17] infers Web object dependencies by injecting delays into the loading of Web objects to deduce the true dependencies between Web objects. *The Mystery Machine* instead leverages a large sample size and the natural variation of timings to infer the causal dependencies between segments. WProf [32] modifies the browser to learn browser page load dependencies. It also injects delays and uses a series of test pages to learn the dependencies and applies a critical path analysis. *The Mystery Machine* looks at end-to-end latency from the server to the client. It automatically deduces a dependency model by analyzing a large set of requests. Google Pagespeed Insight [14] profiles a page load and reports its best estimate of the critical path from the client's perspective. *The Mystery Machine* traces a Web request from the server through the client, enabling it to deduce the end-to-end critical path.

Chen et al. [11] analyzed end-to-end latency of a search service. They also analyzed variation along the server, network, and client components. *The Mystery Machine* analyzes end-to-end latency using critical path analysis, which allows for attributing latency to specific components and performing slack analysis.

Many other systems have looked at automatically discovering service dependencies in distributed systems by analyzing network traffic. Orion [10] passively observes network packets and relies on discovering service dependencies by correlating spikes in network delays. *The Mystery Machine* uses a minimum common content tracing infrastructure finds counterexamples to disprove causal relationship dependencies. WISE [29] answers "what-if" questions in CDN configuration. It uses machine learning techniques to derive important features that affect user response time and uses correlation to de-

rive dependencies between these features. Butkiewicz et al. [8] measured which network and client features best predicted Web page load times across thousands of websites. They produced a predictive model from these features across a diverse set of Web pages. *The Mystery Machine* aims to characterize the end-to-end latency in a single complex Web service with a heterogeneous client base and server environment.

The technique of using logs for analysis has been applied to error diagnosis [2, 34, 33] and debugging performance issues [19, 27].

7 Conclusion

It is challenging to understand an end-to-end request in a highly-concurrent, large-scale distributed system. Analyzing performance requires a causal model of the system, which *The Mystery Machine* produces from observations of component logs. *The Mystery Machine* uses a large number of observed request traces in order to validate hypotheses about causal relationships.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Willy Zwaenepoel, for comments that improved this paper. We also thank Claudiu Gheorghe, James Ide, and Okay Zed for their help and support in understanding the Facebook infrastructure. This research was partially supported by NSF awards CNS-1017148 and CNS-1421441. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing NSF, Michigan, Facebook, or the U.S. government.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, October 2003.

[2] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 193–206, Big Sky, MT, October 2009.

[3] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[4] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, August 2007.

[5] Paul Barford and Mark Crovella. Critical path analysis of TCP transactions. In *Proceedings of the ACM Conference on Computer Communications (SIGCOMM)*, Stockholm, Sweden, August/September 2000.

[6] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, June 2013.

[8] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. Understanding website complexity: Measurements, metrics, and implications. In *Internet Measurement Conference (IMC)*, Berlin, Germany, November 2011.

[9] Anupam Chanda, Alan L. Cox, and Willy Zwanepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, Lisboa, Portugal, March 2007.

[10] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

[11] Yingying Chen, Ratul Mahajan, Baskar Sridharan, and Zhi-Li Zhang. A provider-side view of web search response time. In *Proceedings of the 2013 ACM Conference on Computer Communications*, Hong Kong, China, August 2013.

[12] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), February 2001.

[13] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, pages 271–284, Cambridge, MA, April 2007.

[14] Google. Google Pagespeed Insight. `https://developers.google.com/speed/pagespeed/`.

[15] Eric Koskinen and John Jannotti. Borderpatrol: Isolating events for precise black-box tracing. In *Proceedings of the 3rd ACM European Conference on Computer Systems*, April 2008.

[16] Adam Lazur. Building a billion user load balancer. In *Velocity Web Performance and Operations Conference*, Santa Clara, CA, June 2013.

[17] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert Greenberg, and Yi-Min Wang. Webprophet: Automating performance prediction for web services. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, April 2010.

[18] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-dar. Modeling the parallel execution of black-box services. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Portland, OR, June 2011.

[19] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, April 2012.

[20] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, Lombard, IL, April 2013.

[21] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September 2003.

[22] Lenin Ravindranath, Jitendra Padjye, Sharad Agrawal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[23] Lenin Ravindranath, Jitendra Pahye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Farmington, PA, October 2013.

[24] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, pages 115–128, San Jose, CA, May 2006.

[25] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, TX, March 2013.

[26] Ali Ghassan Saidi. *Full-System Critical-Path Analysis and Performance Prediction*. PhD thesis, Department of Computer Science and Engineering, University of Michigan, 2009.

[27] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, pages 43–56, Boston, MA, March 2011.

[28] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google research, 2010.

[29] Mukarram Bin Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. Answering what-if deployment and configuration questions with wise. In *Proceedings of the 2008 ACM Conference on Computer Communications*, Seattle, WA, August 2008.

[30] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive – a warehousing solution over a map-reduce framework. In *35th International Conference on Very Large Data Bases (VLDB)*, Lyon, France, August 2009.

[31] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier Internet services and its applications. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, Banff, AB, June 2005.

[32] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with wprof. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, April 2013.

[33] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[34] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, Pittsburgh, PA, March 2010.

[35] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The HipHop compiler for PHP. *ACM International Conference on Object Oriented Programming Systems, Languages, and Applications*, October 2012.

[36] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, October 2014.

End-to-end Performance Isolation through Virtual Datacenters

Sebastian Angel[⋆], Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, Eno Thereska
Microsoft Research [⋆]*The University of Texas at Austin*

Abstract

The lack of performance isolation in multi-tenant datacenters at appliances like middleboxes and storage servers results in volatile application performance. To insulate tenants, we propose giving them the abstraction of a dedicated *virtual datacenter* (VDC). VDCs encapsulate end-to-end throughput guarantees—specified in a new metric based on virtual request cost—that hold across distributed appliances and the intervening network.

We present *Pulsar*, a system that offers tenants their own VDCs. Pulsar comprises a logically centralized controller that uses new mechanisms to estimate tenants' demands and appliance capacities, and allocates datacenter resources based on flexible policies. These allocations are enforced at end-host hypervisors through multi-resource token buckets that ensure tenants with changing workloads cannot affect others. Pulsar's design does not require changes to applications, guest OSes, or appliances. Through a prototype deployed across 113 VMs, three appliances, and a 40 Gbps network, we show that Pulsar enforces tenants' VDCs while imposing overheads of less than 2% at the data and control plane.

1 Introduction

In recent years, cloud providers have moved from simply offering on-demand compute resources to providing a broad selection of services. For example, Amazon EC2 offers over twenty services including networked storage, monitoring, load balancing, and elastic caching [1]. Small and enterprise datacenters are also part of this trend [56, 59]. These services are often implemented using *appliances*, which include in-network middleboxes like load balancers and end-devices like networked storage servers. Although *tenants* (i.e., customers) can build their applications atop these services, application performance is volatile, primarily due to the lack of isolation at appliances and the connecting network. This lack of isolation hurts providers too—overloaded appliances are more prone to failure [59].

We present *Pulsar*, the first system that enables datacenter operators to offer appliance-based services while ensuring that tenants receive guaranteed end-to-end throughput. Pulsar gives each tenant a *virtual datacenter* (VDC)—an abstraction that affords them the elasticity and convenience of the shared cloud, without relinquishing the performance isolation of a private datacenter. A VDC is composed of virtual machines (VMs), and resources like virtual appliances and a virtual network that are associated with throughput guarantees. These guarantees are independent of tenants' workloads, hold across all VDC resources, and are therefore end-to-end.

Providing the VDC abstraction to tenants presents two main challenges. First, tenants can be bottlenecked at different appliances or network links, and changing workloads can cause these bottlenecks to shift over time (§2.1). Second, resources consumed by a request at an appliance can vary based on request characteristics (type, size, etc.), appliance internals, and simultaneous requests being serviced. For example, an SSD-backed filestore appliance takes disproportionately longer to serve WRITE requests than READ requests (§2.4). This behavior has two implications: (i) the *capacity*, or maximum achievable throughput, of an appliance varies depending on the workload. This is problematic because the amount of appliance resources that can be allocated to tenants becomes a moving target. (ii) Standard metrics for quantifying throughput, like requests/second or bits/second, become inadequate. For example, offering throughput guarantees in request/second, irrespective of the request type, requires the operator to provision the datacenter conservatively based on the costliest request.

Pulsar addresses these challenges and provides the VDC abstraction. It responds to shifting bottlenecks through a logically centralized controller that periodically allocates resources to tenants based on their VDC specifications, demands, and appliance capacities. These allocations are enforced by rate enforcers, found at end-host hypervisors, through a novel multi-resource token bucket (§4.4). Since the actual cost of serving requests can vary, Pulsar charges requests using their *virtual cost*, given in *tokens* (§3). This is a unified metric common to all VDC resources, and hence throughput in Pulsar is measured in tokens/sec. For each appliance, the provider specifies a *virtual cost function* that translates a request into its cost in tokens. This gives tenants a pre-advertised cost model, and allows the provider to offer guarantees that are independent of tenants' workloads without conservative provisioning.

Pulsar's implementation of the VDC abstraction allows the provider to express different resource allocation policies. The provider can offer VDCs with fixed or minimum guarantees. The former gives tenants predictable

performance, while the latter allows them to elastically obtain additional resources. The provider can then allocate spare resources to tenants based on policies that, for example, maximize profit instead of fairness. Additionally, tenants enjoy full control of their VDC resources and can specify their own allocation policies. For example, tenants can give some of their VMs preferential access to an appliance, or can divide their resources fairly.

The flexibility of these policies comes from decomposing the allocation of resources into two steps: (1) a per-tenant allocation step in which tenants receive enough resources to meet their VDC specifications, and (2) a global allocation step in which spare resources are given to tenants with minimum guarantees that have unmet demand (§4.1). For each step, tenants and the provider can choose from existing multi-resource allocation mechanisms [24, 30, 38, 48, 57] to meet a variety of goals (e.g., fairness, utilization, profit maximization).

Overall, this paper makes the following contributions:

- We propose the VDC abstraction, and present the design, implementation, and evaluation of Pulsar.
- We introduce a unified throughput metric based on *virtual request cost*. This makes it tractable for the provider to offer workload-independent guarantees.
- We design controller-based mechanisms to estimate the demand of tenants and the capacity of unmodified appliances for a given workload.
- We design a rate-limiter based on a new multi-resource token bucket that ensures tenants with changing workloads cannot affect other tenants' guarantees.

A key feature of Pulsar is its ease of deployment. Pulsar isolates tenants without requiring any modifications to applications, guest OSes, appliances, or the network. As a proof of concept, we deployed a prototype implementation of Pulsar on a small testbed comprising eleven servers, 113 VMs, and three types of appliances: an SSD-backed filestore, an in-memory key-value store, and an encryption appliance. We show that Pulsar is effective at enforcing tenant VDCs with data and control plane overheads that are under 2% (§6.3). We also find that controller scalability is reasonable: the controller can compute allocations for datacenters with 24K VMs and 200 appliances in 1–5 seconds (§6.4).

2 Motivation and background

Performance interference in shared datacenters is well documented both in the context of the network [33, 46, 55, 70, 71], and of shared appliances like storage servers (filestores, block stores, and key-value stores) [25, 31, 46], load balancers [49], IDSes [20], and software routers [23]. These observations have led to proposals that provide performance isolation across the network [12, 14, 28, 43, 51, 52, 60, 73], storage servers [26, 62, 66, 72], and middleboxes [23]. However, in all cases the focus is either on a single resource (network or storage), or on multiple resources within a single appliance.

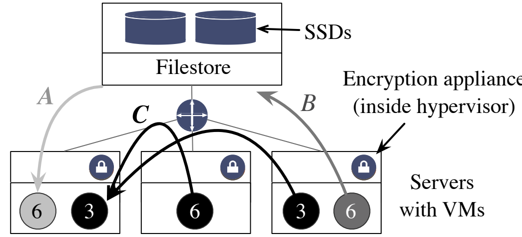
By contrast, today's cloud platforms offer a diverse selection of appliances that tenants can use to compose their applications. Measurements from Azure's datacenters show that up to 44% of their intra-datacenter traffic occurs between VMs and appliances [49]. In this section, we show that tenants can be bottlenecked at any of the appliances or network resources they use, that these bottlenecks can vary over time as tenants' workloads change, and that the end result is variable application performance. Furthermore, we show that existing mechanisms cannot address these challenges.

2.1 How do tenant bottlenecks change?

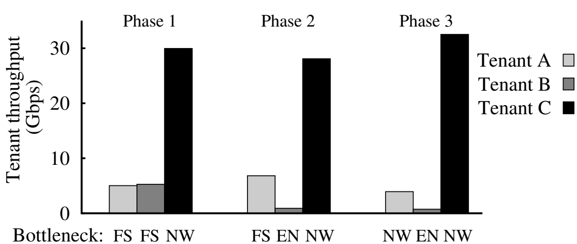
We begin with a simple experiment on our testbed (detailed in Section 6) comprising 16-core servers connected using RDMA over converged Ethernet (RoCE) on a 40 Gbps network. The setup, depicted in Figure 1(a), involves three physical servers and two appliances: a filestore with an SSD back-end and an encryption appliance. The filestore is a centralized appliance providing persistent storage for all the VMs, while the encryption appliance is a distributed appliance present inside the hypervisor at each server. There are three tenants, *A*–*C*, running synthetic workloads on six, six, and twelve VMs respectively. Tenant *A* is reading from the filestore and tenant *B* is writing to the filestore, resulting in 64 KB IO requests across the network. Tenant *C* is running an application that generates an all-to-one workload between its VMs. This models the "aggregate" step of *partition/aggregate* workloads, a common pattern for web applications [10].

We focus on tenant performance across three phases of the experiment—phase transitions correspond to one of the tenants changing its workload. The first set of bars in Figure 1(b) shows the aggregate throughput for the three tenants in phase 1. Tenants *A* and *B* are bottlenecked at the filestore. Having similar workloads, they share the SSD throughput and achieve 5.2 Gbps each. Tenant *C*, with its all-to-one traffic, is bottlenecked at the network link of the destination VM and achieves 29.9 Gbps.

In the next phase, tenant *B*'s traffic is sent through the encryption appliance (running AES). This may be requested by the tenant or could be done to accommodate the provider's security policy. Tenant *B*'s throughput is thus limited by the encryption appliance's internal bottleneck resource: the CPU. As depicted in phase 2 of Figure 1(b), this decreases tenant *B*'s performance by 7×, and has a cascading effect. Since more of the filestore capacity is available to tenant *A*, its performance improves by 36%, thereby reducing tenant *C*'s throughput by 9.2%



(a) Experiment setup with three tenants.



(b) Tenant performance varies as bottlenecks change.

Figure 1—Tenant performance is highly variable and depends on the appliances used and the workloads of other tenants. Numbers in (a) represent the # of VMs for a tenant; arrows represent the direction of traffic. The *x*-axis labels in (b) indicate the bottleneck appliance: "FS" is filestore, "EN" is encryption appliance, and "NW" is network.

	50th	90th	95th	99th	Duration (mins)
Key-value IO	0.04	0.14	0.28	0.41	2
Filestore IO	0.14	0.24	0.32	0.87	2 – 23
Network	0.002	0.004	0.005	0.61	1.3 – 49

Figure 2—Throughput at selected percentiles normalized based on the maximum value in each trace. Large differences between the median and higher percentiles indicate workload changes. The last column shows the duration of these changes.

(since both tenants are colocated on the same server and share the network link).

In phase 3, tenant *C* generates more network flows from each of the source VMs to the destination VM. Since most TCP-like transport protocols achieve per-flow fairness, this allows tenant *C* to grab more of the bandwidth at the destination's network link and its throughput improves. However, this degrades the performance of tenant *A*'s colocated VMs by 2.1×. These VMs are unable to saturate the filestore throughput and are instead bottlenecked at the network.

Overall, these simple yet representative experiments bring out two key takeaways:

- *Variable application performance.* A tenant's performance can vary significantly depending on its workload, the appliances it is using, and the workloads of other tenants sharing these appliances.
- *Multiple bottlenecks.* The performance bottleneck for tenants can vary across space and time. At any instant, tenants can be bottlenecked at different resources across different appliances. Over time, these bottlenecks can vary (as shown by the *x*-label in Fig. 1(b)).

The observations above are predicated on the prevalence of workload changes. We thus study tenant workloads in the context of two production datacenters next.

2.2 How common are workload changes?

We investigate workload changes by examining two traffic traces: (i) a week-long network trace from an enterprise datacenter with 300 servers running over a hundred applications, (ii) a two-day I/O trace from a Hotmail datacenter [67] running several services, including a key-value store and a filestore. Figure 2 tabulates the percentiles of the per-second traffic, normalized to the maximum observed value in each trace. The big difference (orders of magnitude for the key-value and network traces) between the median and higher percentiles indicates a skewed distribution and changing workloads. To study the duration of workload changes, we identified the time intervals where the observed traffic is higher than the 95th percentile. The last column of Figure 2 shows that these workload changes can vary from minutes to almost an hour; such changes are common in both traces.

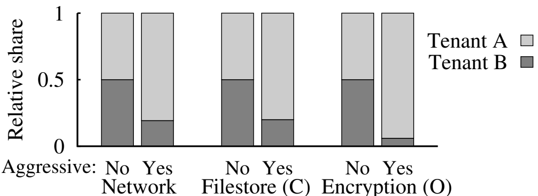


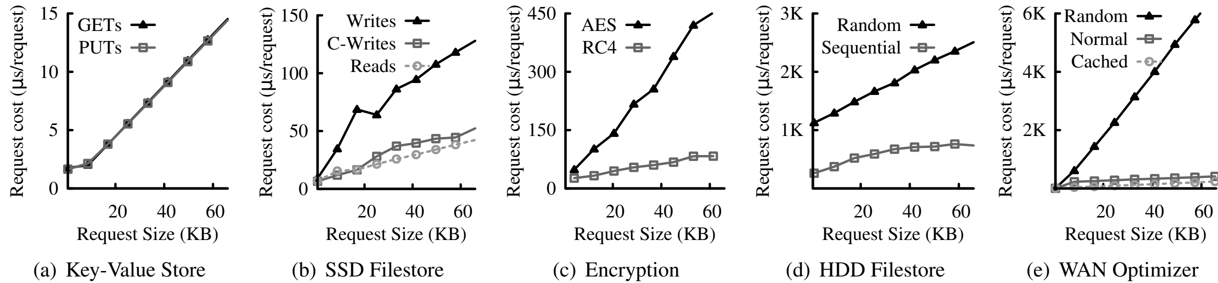
Figure 3—Tenant A can acquire a greater share of any appliance or the network by being aggressive. "(C)" is closed-loop workload, "(O)" is open-loop workload.

2.3 Why is tenant performance affected?

The root cause for variable tenant performance is that neither the network nor appliances isolate tenants from each other. Tenants can even change their workload to improve their performance at others' expense. We expose this behavior through experiments involving two tenants with six VMs each; the results are depicted in Figure 3.

In the first scenario, both tenants generate the same number of TCP flows through a network link, and hence, share it equally. However, tenant *A* can grab a greater share of the link bandwidth simply by generating more flows. For instance, the first set of bars in Figure 3 shows that tenant *A* can acquire 80% of the link bandwidth by generating four times more flows than tenant *B*.

Similar effects can be observed across appliances, but their relative performance depends on the nature of their workload. With closed-loop workloads, each tenant maintains a fixed number of outstanding requests against the appliance. Any tenant can improve its performance



(a) Key-Value Store (b) SSD Filestore (c) Encryption (d) HDD Filestore (e) WAN Optimizer

Figure 4—For appliances, the cost of serving a request can vary with request characteristics. "C-Writes" in (b) represents WRITE requests that can be compressed by the filestore's SSDs. "Sequential" and "Random" in (d) refers to the workload's access pattern (for both READs and WRITEs). (e) depicts the costs of a WAN Optimizer that performs compression for compressible ("Normal") and incompressible ("Random") requests; the cost of serving requests that are cached is also depicted.

by being aggressive and increasing the number of requests outstanding. For example, the second set of bars in Figure 3 shows the relative throughput of two tenants accessing a filestore. When both tenants have the same number of outstanding requests, they share the appliance equally. However, tenant *A* can acquire 80% of the filestore throughput simply by having four times as many requests outstanding as tenant *B*. In the case of open-loop workloads, there is no limit on the number of outstanding requests; in the absence of any isolation at the appliance, a tenant's share is dictated by the transport protocol used. The last set of bars in Figure 3 exposes this behavior.

2.4 Why are absolute guarantees hard to provide?

Isolating appliances is challenging because the resources consumed can vary substantially across individual requests. Figure 4 depicts this observation for five appliances: an in-memory key-value store, an SSD-backed and an HDD-backed filestore, an encryption appliance, and a WAN optimizer that performs compression [8]. For each appliance, we measured its throughput for a stream of requests with identical characteristics. We use the average time for serving a request as an approximation of the *actual request cost*. For the encryption appliance (Fig. 4(c)), the request cost depends on the encryption algorithm being used. For the HDD-backed filestore (Fig. 4(d)), the request cost depends not only on the request size, but also on the access pattern (sequential or random). A request's cost also depends on the appliance internals (including optimizations like caching). For example, Figure 4(b) shows that WRITE requests that can be compressed by the filestore SSDs ("C-Writes") are cheaper to serve than an incompressible write workload.

Another source of variability is the interference between tenant workloads. This exacerbates the difficulty of quantifying a request's cost as a function of its characteristics. The combinatorial explosion resulting from considering all possible workload combinations and the diversity of appliances makes this problem intractable.

Variable request cost has two implications for performance isolation. First, while tenants should ideally receive guarantees in request/sec (or bits/sec) across an appliance, offering such guarantees regardless of tenants' workloads is too restrictive for the provider. Offering guaranteed requests/sec requires provisioning to support tenants always issuing the most expensive request (e.g., maximum-size WRITEs at a filestore appliance). Similarly, offering guaranteed bits/sec requires provisioning based on the request with the maximum cost-to-size ratio. Moreover, both cases require the provider to quantify the actual request cost which, as we discussed, is hard.

The second implication is that the *capacity*, or maximum aggregate throughput, of an appliance can vary over time and across workloads. This is problematic because sharing an appliance in accordance to tenants' guarantees—while ensuring that it is not underutilized—requires a priori knowledge of its capacity.

2.5 Why are existing solutions insufficient?

Existing systems focus on either network or appliance isolation. In Section 6.1, we show that, independently, these solutions do not guarantee end-to-end throughput. This raises a natural question: *is a naive composition of these systems sufficient to provide end-to-end guarantees?* The answer, as we explain below, is no.[1]

Consider a two-tenant scenario in which both tenants are guaranteed half of the available resources. Tenants *A* and *B* each have a single VM sharing a network link and a key-value store appliance (KVS). Tenant *A* issues PUTs and tenant *B* issues GETs to the KVS. On the network, GETs are very small as they contain only the request header; PUTs contain the actual payload. This means that isolating requests based on network semantics (i.e., message size) would allow many more GETs than PUTs to be sent to the KVS. This is problematic because processing GETs at the KVS consumes as many resources as processing PUTs (Fig 4(a)). Even if the KVS optimally

[1] A very similar proposition is discussed as a strawman design in DRFQ [23, §4.2], where each resource within a middlebox is managed by an independent scheduler.

schedules the arriving requests, the task is moot: the scheduler can only choose from the set of requests that actually reaches the KVS. Effectively, tenant *B*'s GETs crowd out tenant *A*'s PUTs, leading to tenant *B* dominating the KVS bandwidth—an undesired outcome!

The key takeaway from this example is that naively composing existing systems is inadequate because their mechanisms are decoupled: they operate independently, lack common request semantics, and have no means to propagate feedback. While it may be possible to achieve end-to-end isolation by bridging network and per-appliance isolation, such a solution would require complex coordination and appliance modifications.

3 Virtual datacenters

We propose virtual datacenters (VDCs) as an abstraction that encapsulates the performance guarantees given to tenants. The abstraction presents to tenants dedicated virtual appliances connected to their VMs through a virtual network switch. Each appliance and VM link is associated with a throughput guarantee that can be either *fixed* or *minimum*. Tenants with fixed guarantees receive the resources specified in their VDCs and no more. Tenants with minimum guarantees forgo total predictability but retain resource elasticity; they may be given resources that exceed their guarantees (when they can use them). These tenants can also specify maximum throughput guarantees to bound their performance variability.

Figure 5 depicts a sample VDC containing two virtual appliances (a filestore and an encryption service), *N* virtual machines, and the connecting virtual network. The guarantees for the filestore and the encryption service are G_s and G_E, respectively. VMs links' guarantees are also depicted. These guarantees are *aggregate*: even if only one or all *N* VMs are accessing the virtual filestore at a given time, the tenant is still guaranteed G_S across it.

For a tenant, the process of specifying a VDC is analogous to that of procuring a small dedicated cluster: it requires specifying the number of VMs, and the type, number, and guarantees of the virtual appliances and virtual network links. Note that VM provisioning (RAM, # cores, etc.) remains unchanged—we rely on existing hypervisors to isolate end-host resources [4, 7, 29, 68]. Providers can offer tenants tools like Cicada [42] and Bazaar [35] to automatically determine the guarantees they need, or tenants can match an existing private datacenter. Alternatively, providers may offer templates for VDCs with different resources and prices, as they do today for VMs (e.g., small, medium, etc.).

Virtual request cost. In Section 2.4 we showed that the actual cost of serving a request at an appliance can vary significantly. We address this by charging requests based on their virtual cost in *tokens*. For each appliance and the network, the provider advertises a *virtual cost function* that maps a request to its cost in tokens. Tenant guarantees across all appliances and the network are thus specified in *tokens/sec*, a unified throughput metric. This strikes a balance between the provider and tenants' requirements. The provider is able to offer workload-independent guarantees without conservative provisioning, while tenants can independently (and statically) determine the requests/second (and bits/second) throughput that can be expected from an appliance.

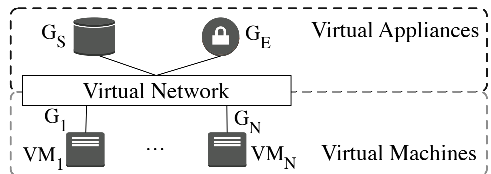


Figure 5—A VDC is composed of virtual appliances and VM links associated with throughput guarantees (in tokens/sec).

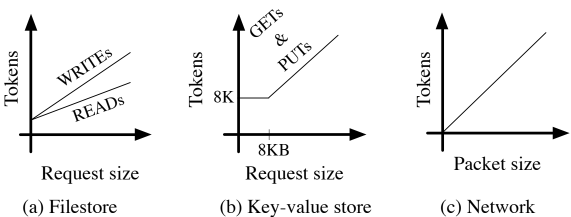


Figure 6—Sample virtual cost functions showing the mapping between request characteristics and their cost in tokens.

Figure 6 shows examples of virtual cost functions. The key-value store cost function states that any request smaller than 8 KB costs a flat 8K tokens, while the cost for larger requests increases linearly with request size. Consider a tenant with a guarantee of 16K tokens/sec. The cost function implies that if the tenant's workload comprises 4 KB PUTs, it is guaranteed 2 PUTs/s, and if it comprises 16 KB PUTs, it is guaranteed 1 PUTs/s. For the network, the relation between packet size and tokens is linear; tokens are equivalent to bytes. Cloud providers already implicitly use such functions: Amazon charges tenants for DynamoDB key-value store requests in integral multiples of 1 KB [2]. This essentially models a virtual cost function with a step-wise linear shape.

The provider needs to determine the virtual cost function for each appliance. This typically involves approximating the actual cost of serving requests through benchmarking, based on historical statistics, or even domain expertise. However, cost functions need not be exact (they can even be deliberately different); our design accounts for any mismatch between the virtual and actual request cost, and ensures full appliance utilization (§4.3). It is thus sufficient for the provider to roughly approximate a request's cost from its observable characteristics. Section 8 discusses appliances for which observable request characteristics are a poor indicator of request cost.

4 Design

Pulsar enables the VDC abstraction by mapping tenant VDCs onto the underlying physical infrastructure and isolating them from each other. Pulsar's architecture, depicted in Figure 7, consists of a logically centralized controller with full visibility of the datacenter topology and tenants' VDC specifications, and a *rate enforcer* inside the hypervisor at each compute server. The controller estimates tenants' demands, appliance capacities, and computes allocations that are sent to rate enforcers. The rate enforcers collect local VM traffic statistics, and enforce tenant allocations.

Pulsar's design does not require the modification of physical appliances, guest OSes, or network elements, which eases the path to deployment. It also reconciles the isolation requirements of tenants with the provider's goal of high utilization by allocating spare capacity to tenants that can use it. Specifically, Pulsar's allocation mechanism achieves the following goals:

G1 *VDC-compliance.* Tenants receive an allocation of resources that meets the guarantees specified in their VDCs. A tenant can choose from different mechanisms to distribute these resources among its VMs.

G2 *VDC-elasticity.* Tenants receive allocations that do not exceed their demands (i.e., the resources they can actually consume). Moreover, spare resources are allocated to tenants with minimum guarantees and unmet demand in accordance to the provider's policy.

4.1 Allocating resources to tenants

We treat each appliance as an atomic black box and do not account for resources inside of it. For example, a key-value store includes internal resources like its CPU and memory, but we treat all of them as a single resource. Henceforth, a "resource" is either a network link or an appliance.[2] Each resource is associated with both a capacity that can vary over time and must be dynamically estimated, and a cost function that maps a request's characteristics into the cost (in tokens) of servicing that request. All of Pulsar's mechanisms act directly on tenant *flows*. A flow encapsulates all connections between a pair of VMs that share the same path (defined in terms of the physical resources used). Note that a flow can have the same source and destination VM, as is the case with flows that access end-devices like storage servers.

Allocations in Pulsar are performed in control intervals (e.g., 1 sec), and involve the controller assigning *allocation vectors* to flows. Each entry in an allocation vector describes the amount of a particular resource that a flow can use over the control interval. A flow's allocation is the sum of two components. First, a *local* component is computed by applying a local policy (chosen by the tenant from a pre-advertised set) to the tenant's VDC. Next, a *global* component is computed by applying a global policy (chosen by the provider) to the physical infrastructure. The local policy describes how a tenant distributes its guaranteed resources to its flows, while the global policy describes how the provider distributes spare resources in the datacenter to flows with unmet demand. We describe multi-resource allocation next, followed by a description of the local and global allocations.

[2]Network links are bidirectional and are treated as two resources.

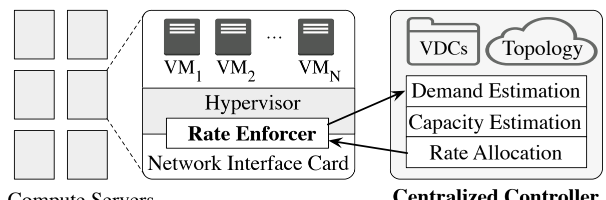


Figure 7—Pulsar's architecture is made up of a centralized controller that apportions resources to tenants, and distributed rate enforcers that uphold these allocations.

Multi-resource allocation (MRA). The goal of an MRA scheme is to distribute multiple *types* of resources among clients with heterogeneous demands. MRA schemes have been around for decades, primarily in the context of multi-capacity (or multi-dimension) bin packing problems [41, 45, 47, 54]. However, recent work [16, 19, 24, 30, 38, 40, 48, 57] has extended MRA schemes to ensure that the resulting allocations are not only efficient, but also meet different notions of fairness.

Generally, an MRA mechanism for m clients (flows in our context) and n resources provides the interface:

$$A \leftarrow MRA(D, W, C) \tag{1}$$

where A, D, and W are $m \times n$ matrices, and C is an n-entry vector. $D_{i,j}$ represents the demand of flow i for resource j, or how much of resource j flow i is capable of consuming in a control interval. $A_{i,j}$ contains the resulting demand-aware allocation (i.e., $A_{i,j} \leq D_{i,j}$ for all i and j). W contains weight entries used to bias allocations to achieve a chosen objective (e.g., weighted fairness, or revenue maximization). C contains the capacity of each resource. With Pulsar, we can plug in any mechanism that implements the interface above for either allocation step.

Local allocations. Pulsar gives each tenant a private VDC. To give tenants control over how their guaranteed resources are assigned to their flows, we allow them to choose a local MRA mechanism (MRA_L). For example, tenants who want to divide their VDC resources fairly across their flows could choose a mechanism that achieves dominant-resource fairness (DRF) [24] or bottleneck-based fairness [19]. Alternatively, tenants may prefer a different point in the fairness-efficiency space, as achieved by other mechanisms [38, 57]. Hence,

tenant t's local allocation matrix (A^t) is given by:

$$A^t \leftarrow MRA_L(D^t, W^t, C^t) \quad (2)$$

D^t and W^t are demand and weight matrices containing only t's flows, and C^t is the capacity vector containing the capacities of each virtual resource in t's VDC. These capacities correspond to the tenant's guarantees, which are static and known a priori (§3). W^t is set to a default (all entries are 1) but can be overridden by the tenant. We describe how flow demands are estimated in Section 4.2.

Global allocations. To achieve VDC-elasticity, Pulsar assigns unused resources to flows with unmet demand based on the provider's global policy.[3] This policy need not be fair or efficient. For example, the provider can choose a global allocation mechanism, MRA_G, that maximizes its revenue by favoring tenants willing to pay more for spare capacity, or prioritizing the allocation of resources that yield a higher profit (even if these allocations are not optimal in terms of fairness or utilization).

The resulting global allocation is the $m \times n$ matrix A^G, where m is the total number of flows (across all tenants with minimum guarantees), and n is the total number of resources in the datacenter. A^G is given by:

$$A^G \leftarrow MRA_G(D^G, W^G, C^G) \quad (3)$$

D^G contains the unmet demand for each flow across each physical resource *after* running the local allocation step; entries for resources that are not in a flow's path are set to 0. The weights in W^G are chosen by the provider, and can be derived from tenants' VDCs to allow spare resources to be shared in proportion to up-front payment (a weighted fair allocation), or set to 1 to allow a fair (payment-agnostic) allocation. The n-entry capacity vector C^G contains the remaining capacity of every physical resource in the datacenter. Since tenants' demands vary over time, we describe how we estimate them next.

4.2 Estimating a flow's demand

The first input to Pulsar's MRA mechanisms is the demand matrix D. A row in D represents the demand vector for a flow, which in turn, contains the demand (in tokens) for each resource along the flow's path. The controller computes each flow's demand vector from estimates provided by rate enforcers. At a high level, the rate enforcer at a flow's source uses old and current request statistics to estimate a flow's demand for the next interval.

A flow's demand is the amount of resources that the application sourcing the flow *could* consume during a control interval, and it depends on whether the application is open- or closed-loop. Open-loop applications have no limit on the number of outstanding requests; the arrival rate is based on external factors like user input or timers. Consequently, a rate enforcer can observe a flow f's demand for the current control interval by tracking both processed and queued requests.

The two components used to estimate the demand for flows of open-loop applications are the *utilization vector* and the *backlog vector*. Flow f's utilization vector, $u_f[i]$, contains the total number of tokens consumed for each resource by f's requests over interval i.[4] Note that if f's requests arrive at a rate exceeding its allocation, some requests will be queued (§4.4). f's backlog vector, $b_f[i]$, contains the tokens needed across each resource in order to process all the requests that are still queued at the end of the interval. Put together, the demand vector for flow f for the next interval, $d_f[i+1]$, is simply the sum of the utilization and backlog vector for the current interval:

$$d_f[i+1] = u_f[i] + b_f[i] \quad (4)$$

Estimating the demand for flows of a closed-loop application is more challenging. These flows maintain a fixed number of outstanding requests which limits the usefulness of the backlog vector (since queuing at any point in time cannot exceed the number of outstanding requests). To address this, we account for queuing that occurs *throughout* a control interval and not just at the end of it. Within each control interval, we obtain periodic samples for the number of requests that are queued above and are outstanding beyond the rate enforcer; a flow's *queuing* (q_f) and *outstanding* (o_f) vectors contain the average number of requests (in tokens) that are queued and outstanding during a control interval. The demand vector for closed-loop flows at interval $i+1$ is thus given by:

$$d_f[i+1] = u_f[i] + q_f[i] \cdot \frac{u_f[i]}{o_f[i]} \quad (5)$$

where "·" and "/" are element-wise operations. The rationale behind the second component is that an average of $o_f[i]$ outstanding tokens results in a utilization of $u_f[i]$. Consequently, if the rate enforcer were to immediately release all queued requests (which on average account for $q_f[i]$ tokens), the maximum expected additional utilization would be: $q_f[i] \cdot (u_f[i]/o_f[i])$.

In practice, however, it is difficult to differentiate between open- and closed-loop workloads. To reduce the probability that our mechanism under-estimates flow demands (which can result in violation of tenants' VDCs), we use the maximum of both equations:

$$d_f[i+1] = u_f[i] + \max\left(b_f[i],\ q_f[i] \cdot \frac{u_f[i]}{o_f[i]}\right) \quad (6)$$

During every control interval, rate enforcers compute and send demand vectors for their flows to the controller, allowing it to construct the per-tenant and the global demand matrices. To avoid over-reacting to bursty workloads, the controller smoothens these estimates through an exponentially weighted moving average.

[3] Tenants with fixed guarantees are excluded from global allocations.

[4] Rate enforcers derive tokens consumed by a flow's requests on resources along its path by applying the corresponding cost functions.

4.3 Estimating appliance capacity

Recall that appliance capacity (measured in tokens/second) is the last input to Pulsar's MRA mechanisms (§4.1). If an appliance's virtual cost function perfectly describes the actual cost of serving a request, the appliance capacity is independent of its workload; the capacity is actually constant. However, determining actual request cost is hard, and thus virtual cost functions are likely to be approximate. This means that the capacity of an appliance varies depending on the given workload and needs to be estimated dynamically.

For networks, congestion control protocols implicitly estimate link capacity and distribute it among flows. However, they conflate capacity estimation with resource allocation which limits them to providing only flow-level notions of fairness, and their distributed nature increases complexity and hurts convergence time. Instead, Pulsar's controller serves as a natural coordination point, allowing us to design a centralized algorithm that estimates appliance capacity independently of resource allocation. This decoupling enables tenant-level allocations instead of being restricted to flow-level objectives. Furthermore, global visibility at the controller means that the mechanism is simple—it does not require appliance modification or inter-tenant coordination—yet it is accurate.

The basic idea is to dynamically adapt the *capacity estimate* for the appliance based on congestion signals. "Congestion" indicates that the capacity estimate exceeds the appliance's actual capacity and the appliance is overloaded. We considered both implicit congestion signals like packet loss and latency [22, 37, 74], and explicit signals like ECN [53] and QCN [32]. However, obtaining explicit signals requires burdensome appliance modifications, while implicit signals like packet loss are not universally supported (e.g., networked storage servers cannot drop requests [58]). Indeed, systems like PARDA [26] use latency as the sole signal for estimating system capacity. Nevertheless, latency is a noisy signal, especially when different flows have widely different paths.[5] Instead, we use the controller's global visibility to derive two implicit congestion signals: *appliance throughput* and *VDC-violation*.

Appliance throughput is the total throughput (in tokens) of all flows across the appliance over a given interval. When the capacity estimate exceeds the actual capacity, the appliance is overloaded and is unable to keep up with its workload. In this case, appliance throughput is less than the capacity estimate, signaling congestion.

The second congestion signal is needed because we aim to determine the appliance's *VDC-compliant capacity*—the highest capacity that meets tenants' VDCs. Since capacity is workload-dependent, and VDCs impact the nature of the workload that reaches the appliance, the VDC-compliant capacity can be lower than the maximum capacity (across all workloads). To understand this, consider a hypothetical encryption appliance that serves either 4 RC4 requests, or 1 RC4 and 1 AES request per second (i.e., AES requests are 3× more expensive than RC4 requests). Assume that the virtual cost function charges 2 tokens for an AES request and 1 token for an RC4 request, and that two tenants are accessing the appliance—t_{AES} with a minimum guarantee of 2 tokens/s, and t_{RC4} with a minimum guarantee of 1 token/s. The VDC-compliant workload in this scenario corresponds to 1 AES and 1 RC4 request every second, resulting in a VDC-compliant capacity of 3 tokens/s. However, notice that the maximum capacity is actually 4 tokens/s (when the workload is 4 RC4 requests). Assuming 1-second discrete timesteps and FIFO scheduling at the appliance, using a capacity of 4 tokens/s in Pulsar's global allocation step would result in t_{AES}'s guarantee being violated at least once every 3 seconds: Pulsar allows an additional RC4 request to go through, leading to uneven queuing at the appliance, and a workload that is not VDC-compliant. To avoid this, we use VDC-violation as a congestion signal.

Capacity estimation algorithm. We use a window-based approach for estimating appliance capacity. At a high level, the controller maintains a probing window in which the appliance's actual capacity (C_{REAL}) is expected to lie. The probing window is characterized by its extremes, $minW$ and $maxW$, and is constantly refined in response to the presence or absence of congestion signals. The current capacity estimate (C_{EST}) is always within the probing window and is used by the controller for rate allocation. The refinement of the probing window comprises four phases:

① *Binary search increase.* In the absence of congestion, the controller increases the capacity estimate to the midpoint of the probing window. This binary search is analogous to BIC-TCP [74]. The controller also increases $minW$ to the previous capacity estimate as a lack of congestion implies that the appliance's actual capacity exceeds the previous estimate. This process repeats until stability is reached or congestion is detected.

② *Revert.* When congestion is detected, the controller's response depends on the congestion signal. On observing the throughput congestion signal, the controller reverts the capacity estimate to $minW$. This ensures that the appliance does not receive an overloading workload for more than one control interval. Further, $maxW$ is reduced to the previous capacity estimate since the appliance's actual capacity is less than this estimate.

③ *Wait.* On observing the VDC-violation signal, the controller goes through the revert phase onto the wait phase. The capacity estimate, set to $minW$ in the revert

[5] PARDA dampens noise through inter-client coordination.

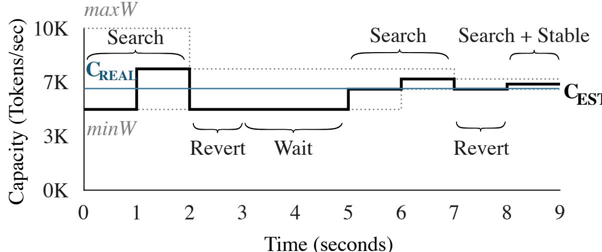


Figure 8—Pulsar estimates an appliance's capacity by probing for a higher capacity and responding to congestion signals. Stability is reached once the probing window is small enough.

phase, is not changed until all guarantees are met again. This allows the appliance, which had been overloaded earlier, to serve all outstanding requests. This is particularly important as, unlike network switches, many appliances cannot drop requests.

④ *Stable.* Once the probing window is small enough (e.g., 1% of the maximum capacity), the controller reaches the stable state in which the capacity estimate is adjusted in response to minor fluctuations in workload. Our mechanism relies on tracking the average number of outstanding requests (measured in tokens) at the appliance during interval i, $O[i]$,[6] and comparing its value to the average number of outstanding requests at the appliance at the beginning of the stable phase, O_s. The difference between these observations affects C_{EST} as follows:

$$C_{EST}[i+1] = C_{EST}[i] - \alpha \cdot (O[i] - O_s) \qquad (7)$$

where α governs the sensitivity to workload changes. The rationale is that O_s serves as a good predictor of the number of outstanding requests that can be handled by the appliance when it is the bottleneck resource. When the outstanding requests at interval i ($O[i]$) exceed this amount, the appliance is likely to be overloaded; the estimate is reduced to ensure that fewer requests are let through by the rate enforcers during the next interval. The opposite is also true. Furthermore, workloads reaching the appliance that differ significantly (more than 10%) from the workload at the beginning of the stable phase restart the estimation process.

Figure 8 depicts a sample run of our algorithm on an appliance with an actual capacity (C_{REAL}) of 6500 tokens/second. Both *minW* and *maxW* are initialized to conservative values known to be much lower/higher than C_{REAL}, while the current estimate (C_{EST}) is initialized to *minW*. The dotted lines represent *minW* and *maxW*, while the solid line represents C_{EST}. At time $t = 1$, there is no congestion signal, so the controller enters phase ①, resulting in C_{EST} being set to 7500 (an overestimate). During the next interval, the controller notices that the total appliance throughput does not match C_{EST}, which triggers phase ②. The queues that built up at the appliance due to capacity overestimation remain past $t = 3$, causing VDCs to be violated and leading into phase ③. This lasts until time $t = 5$, at which point all remnant queues have been cleared and the controller is able to go back to phase ①. This process repeats until stability is reached at time $t = 9$.

[6]The average number of outstanding requests at an appliance, O, is derived by summing over all flows' outstanding vectors (§4.2) and retrieving the entry corresponding to the appliance.

4.4 Rate enforcement

Pulsar rate limits each flow via a rate enforcer found at the flow's source hypervisor. Existing single-resource isolation systems use token buckets [65, §5.4.2] to rate-limit flows. However, traditional token buckets are insufficient to enforce multi-resource allocations, as tenants with changing workloads can consume more resources than they are allocated.

To understand why, assume a rate-limiter based on a single-resource token bucket where the bucket is filled with tokens from the first entry in a flow f's allocation vector (the same applies to any other entry). Further assume that f goes through 2 resources and its estimated demand vector at interval i is $\langle 800, 5000\rangle$ (i.e., f is expected to use 800 tokens of resource 1 and 5,000 of resource 2). Suppose that the controller allocates to f all of its demand, and hence f's allocation vector is also $\langle 800, 5000\rangle$. If f changes its workload and its actual demand is $\langle 800, 40000\rangle$—e.g., a storage flow switching from issuing ten 500 B READs to ten 4 KB READs, where the request messages are the same size but the response message size increases—the rate-limiter would still let ten requests go through. This would allow f to consume 8× its allocation on resource 2; an incorrect outcome!

To address this, we propose a *multi-resource token bucket* that associates multiple buckets with each flow, one for each resource in a flow's path. Each bucket is replenished at a rate given by the flow's allocation for the corresponding resource. For example, in the above scenario, a request is let through only if each bucket contains enough tokens to serve the request. Since f was allocated 5,000 tokens for resource 2, only one 4 KB READ is sent during interval i, and the remaining requests are queued until enough tokens are available. This mechanism ensures that even if a flow's workload changes, its throughput over the next control interval cannot exceed its allocation, and thus cannot negatively impact the performance of other flows or tenants.

4.5 Admission control

Pulsar's allocation assumes that tenants have been admitted into the datacenter, and their VDCs have been mapped onto the physical topology in a way that ensures that enough physical resources are available to meet their guarantees. This involves placing VMs on physical

servers and virtual appliances on their respective counterparts. While VM placement is well-studied [12, 14, 18, 44, 50, 73], prior proposals do not consider appliance placement. Our observation is that Hadrian's [14] placement algorithm can be adapted to support appliances.

Hadrian proposes a technique for modeling network bandwidth guarantees (among a tenant's VMs and across tenants) as a max-flow network problem [13, §4.1.1]. The result is a set of constraints that guide VM placement in the datacenter. Pulsar's VDCs can be similarly modeled. The key idea is to treat appliances as "tenants" with a single VM, and treat all VM-appliance interactions as communication between tenants. Consequently, we are able to model the guarantees in tenants' VDCs as a maximum-flow network, derive constraints for both VM and virtual appliance placement, and use Hadrian's greedy placement heuristic to map tenants' VDCs.

However, Hadrian's placement algorithm requires that the minimum capacity of each resource be known at admission time. While we assume that both the datacenter topology and link capacities are static and well known, determining the minimum capacity for each appliance is admittedly burdensome. Fortunately, Libra [61] proposes a methodology that, while tailored to SSDs, is general enough to cover numerous appliances. Furthermore, the work needed to derive appliances' minimum capacities can be used towards deriving cost functions as well.

5 Implementation

We implemented a Pulsar prototype comprising a standalone controller and a rate enforcer. The rate enforcer is implemented as a filter driver in Windows Hyper-V. There are two benefits from a hypervisor-based implementation. First, Pulsar can be used with unmodified applications and guest OSes. Second, the hypervisor contains the right semantic context for understanding characteristics of requests from VMs to appliances. Thus, the rate enforcer can inspect the header for each request to determine its cost. For example, for a request to a key-value appliance, the enforcer determines its type (GET/PUT) and its size in each direction (i.e., from the VM to the appliance and back). For encryption requests, it determines the request size and kind of encryption.

The driver implementing the rate enforcer is ≈11K lines of C; 3.1K for queuing and request classification, 6.8K for stat collection, support code, and controller communication, and 1.1K for multi-resource token buckets. The rate enforcer communicates with the controller through a user-level proxy that uses TCP-based RPCs; it provides demand estimates to the controller, and receives information about flows' paths, cost functions, and allocations. Each flow is associated with a multi-resource token bucket. The size of each bucket is set to a default of token rate $\times$ 1 second. A 10 ms timer refills tokens and determines the queuing and outstanding vectors (§4.2).

The controller is written in ≈6K lines of C# and runs on a separate server. Inputs to the controller include a topology map of the datacenter, appliances' cost functions, and tenants' VDC specifications. The control interval is configurable and is set to a default of 1 second. Our traces show that workload changes often last much longer (§2.1), so a 1 second control interval ensures good responsiveness and stresses scalability. The controller estimates appliance capacity as described in Section 4.3. To prevent reacting to spurious congestion signals that result from noisy measurements we require multiple consistent readings (3 in our experiments).

At the controller, we have implemented DRF [24], H-DRF [16], and a simple first-fit heuristic as the available MRA mechanisms. In our experiments, we use DRF for local allocations (all weights are set to 1), and H-DRF for global allocations (weights are derived from tenants' guarantees). When computing these allocations the controller sets aside a small amount of headroom (2–5%) across network links. This is used to allocate each VM a (small) default rate for new VM-to-VM flows, which enables these flows to ramp up while the controller is contacted. Note that a new TCP connection between VMs is not necessarily a new flow since all transport connections between a pair of VMs (or between a VM and an appliance) are considered as one flow (§4).

6 Experimental evaluation

To evaluate Pulsar we use a testbed deployment coupled with simulations. Our testbed consists of eleven servers, each with 16 Intel 2.4 GHz cores and 380 GB of RAM. Each server is connected to a Mellanox switch through a 40 Gbps RDMA-capable Mellanox NIC. At the link layer, we use RDMA over converged Ethernet (RoCE). The servers run Windows Server 2012 R2 with Hyper-V as the hypervisor and each can support up to 12 VMs. We use three appliances: (i) a filestore with 6 SSDs (Intel 520) as the back-end, (ii) an in-memory key-value store, and (iii) an encryption appliance inside the hypervisor at each server. Admission control and placement is done manually. Overall, our key findings are:

- Using trace-driven experiments, we show that Pulsar can enforce tenants' VDCs. By contrast, existing solutions do not ensure end-to-end isolation.
- Our capacity estimation algorithm is able to predict the capacity of appliances over varying workloads.
- We find that Pulsar imposes reasonable overheads at the data and control plane. Through simulations, we show that the controller can compute allocations of rich policies for up to 24K VMs within 1-5 seconds.

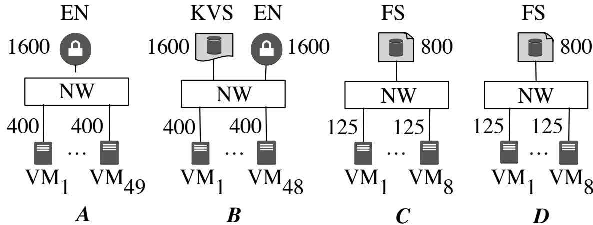


Figure 9—VDCs with minimum guarantees for tenants *A*–*D*

Tenant	READ or GET %	IO Size	Outstanding IOs #
Database IO (*B*)	61%	8 KB	8
Trans. Log (*C*)	1%	0.5 KB	64
Email (*D*)	56%	64 KB	8

Figure 10—Workload characteristics of each VM for tenants *B*–*D* are derived from a two-day Hotmail trace (§2.2).

6.1 Virtual datacenter enforcement

We first show that Pulsar enforces tenants' VDCs. The experiment involves four tenants, *A*–*D*, with their VDCs shown in Figure 9. For example, tenant *A* has forty-nine VMs, each with a minimum network bandwidth of 400 MT/s. For the network, tokens are equivalent to bytes, so these VMs have a guarantee of 400 MB/s (3.2 Gbps). This tenant also has a virtual encryption appliance with a minimum guarantee of 1600 MT/s. For ease of exposition we use the same cost function for all three appliances in our testbed: small requests ($\leq$8 KB) are charged 8 Ktokens, while the cost for other requests is the equivalent of their size in tokens (Figure 6(b) depicts the shape of this cost function).

Workloads. Tenant *A* has an all-to-one workload with its VMs sending traffic to one destination VM (this models a partition/aggregate workflow [10]). We use Iometer [5] parameterized by Hotmail IO traces (§2.2) to drive the workloads for tenants *B*–*D*; we tabulate their characteristics in Figure 10. Database IO is used for tenant *B*'s key-value store access, while Transactional log IO and Email message IO to Hotmail storage are used for *C* and *D* respectively. Traffic from tenants *A* and *B* is encrypted with RC4 by the encryption appliance before being sent on the wire. Since tenant *C* generates 512 byte requests that cost 8 Ktokens, its bytes/sec throughput is $\frac{1}{16}^{th}$ of the reported tokens/sec. The bytes/sec throughput for other tenants is the same as their tokens/sec.

Tenants *A* and *C* are aggressive: each of *A*'s VMs has 8 connections to the destination, while *C*'s generate a closed-loop workload with 64 outstanding IO requests.

Topology. Figure 11 shows the physical topology of the testbed. We arrange tenants' VMs and appliances across our servers so that at least two tenants compete for each resource. Tenants *A* and *B* compete for the encryption appliance. Tenants *C* and *D* compete for the bandwidth at the physical filestore appliance. Further, the destination

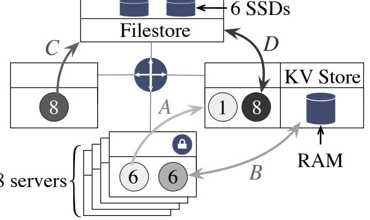


Figure 11—Testbed's physical topology. Numbers indicate # of VMs while arrows show the direction of traffic.

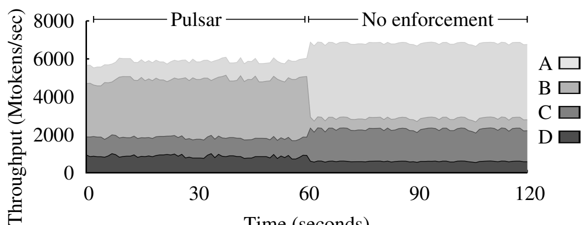


Figure 12—VM-to-VM and VM-to-appliance traffic of four tenants (113 VMs). Pulsar ensures all guarantees are met.

VM for tenant *A*, the key-value store used by tenant *B*, and all the VMs of tenant *D* are co-located on the same server. Thus, they compete at the server's network link.

Tenant guarantees. For the workload in this experiment, the end-to-end throughput for tenants *A*-*D* should be at least 400, 1600, 800, 800 MT/s respectively.

Tenant performance. The first half of Figure 12 shows that, with Pulsar, the aggregate throughput of each tenant exceeds its minimum guarantee. Further, spare capacity at each resource is shared in proportion to tenants' guarantees. On average, tenant *A* gets 1100 MT/s for its VM-to-VM traffic, tenant *B* gets 3150 MT/s for its key-value traffic, and tenants *C* and *D* get 930 and 900 MT/s across the filestore respectively.

By contrast, the second half of Figure 12 shows baseline tenant throughput without Pulsar. We find that the aggressive tenants (*A* and *C*) are able to dominate the throughput of the underlying resources at the expense of others. For instance, tenants *C* and *D* have the same guarantee to the filestore but *C*'s throughput is 3$\times$ that of *D*'s. Tenant *B*'s average throughput is just 580 MT/s, 64% lower than its guarantee. Similarly, tenant *D*'s average throughput is 575 MT/s, 28% lower than its guarantee.

In this experiment, the total throughput (across all tenants) with Pulsar is lower than without it by 8.7%. This is because Pulsar, by enforcing tenant guarantees, is effectively changing the workload being served by the datacenter. Depending on the scenario and the cost functions, such a workload change can cause the total throughput to either increase or decrease (with respect to the baseline).

We also experimented with prior solutions for single-resource isolation. DRFQ [23] achieves per-appliance isolation for general middleboxes, while Pisces [62] and

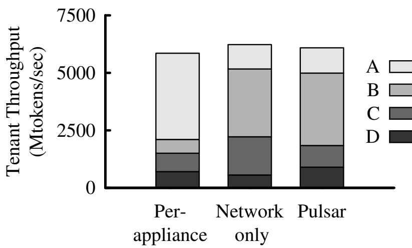
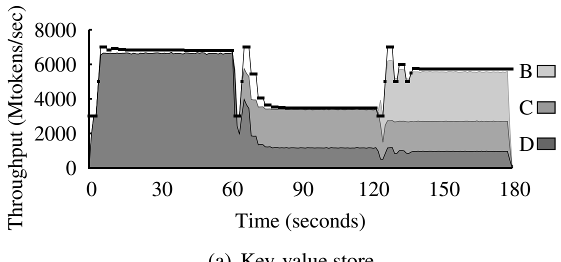
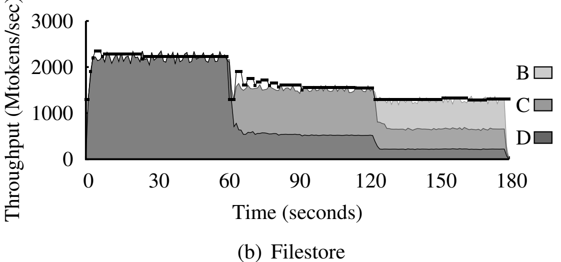


Figure 13—Prior isolation mechanisms fail to meet tenants' guarantees. Per-appliance isolation violates C and D's guarantee, while network only isolation violates D's guarantee.



(a) Key-value store



(b) Filestore

Figure 14—Pulsar's capacity estimation. The solid black line represents the estimated capacity. The guarantees of the three tenants have a ratio of 3:2:1 which is preserved throughout.

IOFlow [66] focus on storage appliances. With such per-appliance isolation, the filestore, key-value store, and encryption appliances are shared in proportion to tenant guarantees but not the network. Figure 13 shows that with per-appliance isolation, tenants *B* and *D* miss their guarantees by 63% and 12% respectively. Note that even though tenant *D* is bottlenecked at the filestore, it is sharing network links with tenant *A* whose aggressiveness hurts *D*'s performance. We also compare against network-only isolation, as achieved by Hadrian [14]. Figure 13 shows that with this approach, tenant *C* is still able to hog the filestore bandwidth at the expense of tenant *D*, resulting in *D*'s guarantee being violated by 30%.

6.2 Capacity estimation

We evaluate Pulsar's capacity estimation algorithm with an experiment that involves three tenants, *B–D*, whose workloads are tabulated in Figure 10. Unlike the previous experiments, we focus on one appliance at a time, and change the setup so that all three tenants use the appliance being evaluated. The appliance guarantees for the tenants are 600, 400, and 200 MT/s, respectively.

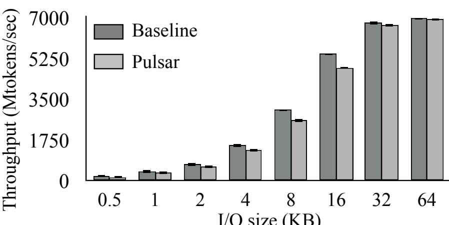


Figure 15—Baseline (no rate limiting) and Pulsar's throughput

To experiment with varying workloads, we activate tenants one at a time. Figure 14(a) shows the estimated capacity and tenant throughput for the key-value store appliance. In the first phase, tenant *D* operates in isolation. The capacity estimate starts at a low value (3000 MT/s) and increases through the binary search phase until the appliance is fully utilized. After 8 seconds, the capacity estimate stabilizes at 6840 MT/s. Tenant *C* is activated next. Its VMs generate small 512 B requests that are more expensive for the key-value store to serve than they are charged, so the appliance's capacity reduces. The controller detects this workload change and the capacity estimate is reduced until it stabilizes at 3485 MT/s. Finally, when tenant *B* is activated, the appliance's actual capacity increases as the fraction of small requests (from *C*'s VMs) reduces. The controller searches for the increased capacity and the estimate stabilizes at 5737 MT/s. Note that the guarantees of all three tenants are met throughout. Using H-DRF as the MRA_G mechanism ensures that spare resources are given based on tenants' guarantees, preserving the 3:2:1 ratio. In all three phases, the estimate converges within 15 seconds.

Figure 14(b) shows capacity estimation for the filestore. As tenants are added, their workloads increase the percentage of small WRITEs, leading to a decrease in the appliance's capacity. The root cause for the lower capacity is that the cost function that we chose undercharges all small requests and incorrectly charges WRITEs the same as READs (cf. Fig 4(b)). To account for this mismatch, the capacity estimate is consistently refined and converges to a value that ensures the appliance is neither being underutilized nor are tenants' guarantees being violated. We validate our observations by re-running the experiments with more accurate cost functions. The result is a capacity estimate that remains constant despite workload changes. We also experimented with the encryption appliance and the HDD-filestore, and the estimation results are similar. We omit them for brevity.

6.3 Data- and control-plane overheads

We first quantify the data-plane overhead of our rate enforcer. We measure the throughput at an unmodified Hyper-V server and compare it to the throughput when

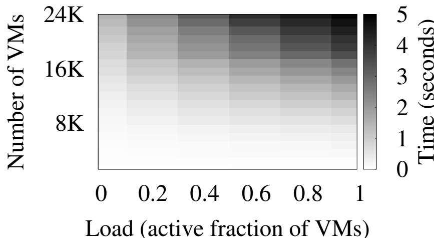


Figure 16—Average time for computing rate allocations

our rate limiter is enabled. To show worst-case overheads we use the in-memory key-value store since that achieves the highest baseline throughput. 12 VMs are used to generate a workload with the same number of PUTs and GETs. We vary the request size from 512 B to 64 KB, thus shifting the bottleneck from the key-value store's CPU for small IO requests to the network for larger requests.

Figure 15 shows the average throughput from 5 runs. The worst-case reduction in throughput is 15% and happens for small request sizes (<32 KB). This overhead is due mostly to data structure locking at high 40+ Gbps speeds. The overhead for requests larger than 32 KB is less than 2%. The CPU overhead at the hypervisor was less than 2% in all cases.

In terms of control-plane overhead, the network cost of the controller updating rate allocations at the servers is 140 bytes/flow, while the cost of transmitting statistics to the controller is 256 bytes/flow per control interval. For example, for 10,000 flows this would mean 10.4 Mbps of traffic from the controller to the rate limiters and 19.52 Mbps of traffic to the controller. Both numbers are worst-case—if the rate allocation or the statistics collected by the rate enforcer do not change from one interval to the next, then no communication is necessary. The latency (including work done by both the controller and the hypervisor) for setting up a rate limiter for a given flow is approximately 83 μs. In general, these numbers indicate reasonable control plane overheads.

6.4 Controller scalability

We evaluate controller scalability through large-scale simulations. Flow demand estimation and appliance capacity estimation incur negligible costs for the controller. Local allocations are parallelizable, and involve much fewer flows and resources than global allocations. We thus focus on quantifying the cost of computing global allocations. We simulate a datacenter with a fat-tree topology [9] and 12 VMs per physical server. Tenants are modeled as a set of VMs with a VDC specification. Each VM sources one flow, either to another VM, or to an appliance. This results in 12 flows per server, which is twice the average number observed in practice [39, §4.1]. Based on recent datacenter measurements [49, Fig. 3], we configure 44% of flows to go to appliances while the rest are VM-to-VM flows. For resources, we model each server's network link (uplink and downlink) and all physical appliances. Thus, for a datacenter with 2000 servers and 200 appliances, we model 4200 resources.

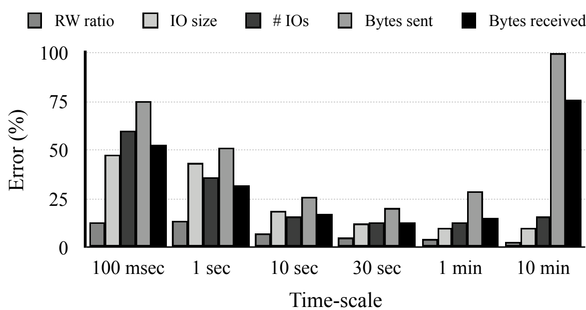


Figure 17—The choice of control interval affects the accuracy of the utilization vector (§4.2) for estimating future demand. The error reduces when the control interval is ≈10–30 seconds.

Figure 16 shows the average allocation time with our iterative DRF implementation as the global allocation mechanism; we vary the total number of VMs and the fraction of VMs that are active. We find that our controller can compute allocations for a medium-scale datacenter with 24K VMs and 2000 servers within 1–5 seconds. When only 20% of the VMs are active, allocation time is at or below 1.5 seconds, even with 24K VMs. However, high loads can push allocation time to as high as 4.9 seconds. The majority of the time is spent performing element-wise arithmetic operations on the 4200-entry vectors. This suggests that parallelizing these operations could provide meaningful performance benefits. We are currently experimenting with GPU implementations of different allocation mechanisms.

Repeating the experiment with H-DRF shows that costs are an order of magnitude higher. This highlights the tradeoff between a policy's expressiveness and its computational burden.

6.5 Choice of control interval

Resource allocation in Pulsar is demand-driven. Hence, the ideal control interval should capture the true demand of flows. Estimating demand for a very short future interval can be impacted by bursts in workload while estimating for a long interval may not be responsive enough (i.e., it may not capture actual workload changes). To verify this, we used our network and IO traces (§2.2) to estimate flow demand at various time-scales. Unfortunately, the traces do not have queuing and backlog information, so we cannot use Pulsar's demand estimation mechanism detailed in Section 4.2. Instead, we simply use past utilization as an indicator of future demand. Specifically, we approximate the demand for a time interval using an exponentially weighted moving average of the utilization over the previous intervals. Thus, the demand errors we report are an over-estimate.

For the network, we use two demand metrics: bytes sent and received. For the storage traffic, the metrics are the number and mean size of IOs, and the read-to-write IO ratio. Figure 17 shows the average demand estimation error across several time-scales. As is well-known [15, 39], most workloads exhibit bursty behavior at very fine timescales (below 1 sec); hence, using a very short control interval leads to large estimation errors. At large time scales (several minutes), past utilization is a poor predictor of future demand. For these workloads, a control interval of ≈10–30 seconds is best suited for demand estimation, offering a good trade-off between responsiveness and stability. While preliminary, these results indicate that Pulsar's controller-based architecture can cope with real datacenter workloads.

7 Related work

Section 2 briefly described existing work on inter-tenant performance isolation. Below we expand that description and contrast related work to Pulsar.

Appliance isolation. A large body of recent work focuses on storage isolation [17, 26, 27, 62, 66], but in all cases the network is assumed to be over-provisioned. While DRFQ [23] achieves fair sharing of multiple resources within a single appliance, it differs from Pulsar mechanistically and in scope: Pulsar decouples capacity estimation from resource allocation, and provides isolation across multiple appliances and the network. Furthermore, Pulsar provides workload-independent guarantees by leveraging an appliance-agnostic throughput metric.

Like Pisces [62] and IOFlow [66], Pulsar uses a centralized controller to offer per-tenant guarantees. However, Pisces relies on IO scheduling at the storage server, while Pulsar performs end-host enforcement without appliance modification. Moreover, Pulsar dynamically estimates the capacity of appliances, whereas IOFlow requires that they be known a priori.

Network isolation. Numerous systems isolate tenants across a shared datacenter network [11, 12, 14, 28, 36, 43, 51, 52, 60, 73]. Beyond weighted sharing [60] and fixed reservations [12, 28, 73], recent efforts ensure minimum network guarantees, both with switch modifications [11, 14, 51], and without them [36, 43, 52]. Pulsar extends the latter body of work by providing guarantees that span datacenter appliances and the network.

Market-based resource pricing. Many proposals allocate resources to bidding users based on per-resource market prices that are measured using a common virtual currency [21, 34, 63, 64, 69]. However, the value of a unit of virtual currency in terms of actual throughput (e.g., requests/sec) varies with supply and demand. Consequently, a tenant's throughput is not guaranteed. By contrast, Pulsar charges requests based on their virtual cost (measured in tokens). While tokens can be seen as a virtual currency, the fact that each resource is associated with a pre-advertised virtual cost function means that a tenant's guarantees in tokens/sec can still be statically translated into guarantees in requests/sec.

Virtual Datacenters. The term VDC has been used as a synonym for Infrastructure-as-a-service offerings (i.e., VMs with CPU and memory guarantees [3, 6]). SecondNet [28] extended the term to include network address and performance isolation by associating VMs with private IPs and network throughput guarantees. Pulsar broadens the VDC definition to include appliances and ensures throughput guarantees across all resources.

8 Discussion and summary

Pulsar's design relies on cost functions that translate requests into their virtual cost. However, for some appliances, observable request characteristics (size, type, etc.) are not a good indicator of request cost. For example, quantifying the cost of a query to a SQL database requires understanding the structure of the query, the data being queried, and database internals. Similarly, the isolation of appliances that perform caching requires further work. While Pulsar implicitly accounts for caching through higher capacity estimates, it does not discriminate between requests that hit the cache and those that do not. We are experimenting with *stateful cost functions* that can charge requests based on past events (e.g., repeated requests within an interval cost less), to explicitly account for such appliances.

In summary, Pulsar gives tenants the abstraction of a virtual datacenter (VDC) that affords them the performance stability of a in-house cluster, and the convenience and elasticity of the shared cloud. It uses a centralized controller to enforce end-to-end throughput guarantees that span multiple appliances and the network. This design also allows for a simple capacity estimation mechanism that is both effective, and appliance-agnostic. Our prototype shows that Pulsar can enforce tenant VDCs with reasonable overheads, and allows providers to regain control over how their datacenter is utilized.

Acknowledgments

This paper was improved by conversations with Attilio Mainetti, Ian Kash, Jake Oshins, Jim Pinkerton, Antony Rowstron, Tom Talpey, and Michael Walfish; and by helpful comments from Josh Leners, Srinath Setty, Riad Wahby, Edmund L. Wong, the anonymous reviewers, and our shepherd, Remzi Arpaci-Dusseau. We are also grateful to Swaroop Kavalanekar and Bruce Worthington for the Hotmail IO traces, Andy Slowey for testbed support, and Mellanox for the testbed switches. Sebastian Angel was supported by NSF grant 1040083 during the preparation of this paper.

References

[1] Amazon AWS Products. http://aws.amazon.com/products/.

[2] Amazon DynamoDB capacity unit. http://aws.amazon.com/dynamodb/faqs/#What_is_a_readwrite_capacity_unit.

[3] Bluelock: Virtual data centers. http://www.bluelock.com/virtual-datacenters.

[4] Hyper-V resource allocation. http://technet.microsoft.com/en-us/library/cc742470.aspx.

[5] Iometer. http://iometer.org.

[6] VMware vCloud: Organization virtual data center. http://kb.vmware.com/kb/1026320.

[7] vSphere resource management guide. https://www.vmware.com/support/pubs/vsphere-esxi-vcenter-server-pubs.html.

[8] WANProxy. http://wanproxy.org.

[9] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2008.

[10] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2010.

[11] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. The price is right: Towards location-independent costs in datacenters. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2011.

[12] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2011.

[13] H. Ballani, D. Gunawardena, and T. Karagiannis. Network sharing in multi-tenant datacenters. Technical Report MSR-TR-2012-39, MSR, 2012.

[14] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawaradena, and G. O'Shea. Chatty tenants and the cloud network sharing problem. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2013.

[15] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. In *Proceedings of the ACM SIGCOMM Workshop on Research on Enterprise Networking (WREN)*, Aug. 2009.

[16] A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, Oct. 2013.

[17] J.-P. Billaud and A. Gulati. hClock: Hierarchical QoS for packet scheduling in a hypervisor. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2013.

[18] O. Biran, A. Corradi, M. Fanelli, L. Foschini, A. Nus, D. Raz, and E. Silvera. A stable network-aware VM placement for cloud systems. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2012.

[19] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial. No justified complaints: On fair sharing of multiple resources. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, Aug. 2012.

[20] H. Dreger, A. Feldman, V. Paxson, and R. Sommer. Predicting the resource consumption of network intrusion detection systems. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Sept. 2008.

[21] M. Feldman, K. Lai, and L. Zhang. A price-anticipating resource allocation mechanism for distributed shared clusters. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, June 2005.

[22] S. Floyd. HighSpeed TCP for large congestion windows, Dec. 2003. RFC 3649. http://www.ietf.org/rfc/rfc3649.txt.

[23] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2012.

[24] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[25] D. Ghoshal, R. S. Canon, and L. Ramakrishnan. I/O performance of virtualized cloud environments. In *Proceedings of the International Workshop on Data Intensive Computing in the Clouds (DataCloud)*, May 2011.

[26] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: proportional allocation of resources for distributed storage access. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2009.

[27] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling throughput variability for hypervisor IO scheduling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[28] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Nov. 2010.

[29] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in XEN. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, Dec. 2006.

[30] A. Gutman and N. Nisan. Fair allocation without trade. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, June 2012.

[31] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of Windows Azure. In *Proceedings of the International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, June 2010.

[32] IEEE Computer Society. Virtual bridged local area networks. Amendment 13: Congestion notification, Apr. 2010. IEEE Std. 802.1Qau-2010.

[33] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of cloud services. Technical Report PDS-2010-002, Delft University, 2010.

[34] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-recharging virtual currency. In *Proceedings of the ACM SIGCOMM Workshop on the Economics of Peer-to-Peer Systems (P2PECON)*, Aug. 2005.

[35] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2012.

[36] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical network performance isolation at the edge. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2013.

[37] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: Motivation, architecture, algorithms, performance. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 2004.

[38] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2012.

[39] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, Nov. 2009.

[40] I. Kash, A. D. Procaccia, and N. Shah. No agent left behind: Dynamic fair division of multiple resources. In *Proceedings of the*

International Conference on Autonomous Agents and Multiagent Systems (AAMAS), May 2013.

[41] L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. *IBM Journal of Research and Development*, 21(5), Sept. 1977.

[42] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2014.

[43] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banarjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2014.

[44] S. Lee, R. Panigraphy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder. Validating heuristics for virtual machines consolidation. Technical Report MSR-TR-2011-9, Microsoft Research, 2011.

[45] W. Leinberger, G. Karypis, and V. Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Sept. 1999.

[46] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, Nov. 2010.

[47] K. Maruyama, S. K. Chang, and D. T. Tang. A general packing algorithm for multidimensional resource requirements. *International Journal of Computer and Information Sciences*, 6(2), 1977.

[48] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, June 2012.

[49] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2013.

[50] J. T. Piao and J. Yan. A network-aware virtual machine placement and migration approach in cloud computing. In *Proceedings of the International Conference on Grid and Cloud Computing (GCC)*, Nov. 2010.

[51] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2012.

[52] L. Popa, P. Yalagandula, S. Banarjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2013.

[53] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP, Sept. 2001. RFC 3168. `http://www.ietf.org/rfc/rfc3168.txt`.

[54] N. Roy, J. S. Kinnebrew, N. Shankaran, G. Biswas, and D. C. Schmidt. Toward effective multi-capacity resource allocation in distributed real-time embedded systems. In *Proceedings of the IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*, May 2008.

[55] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Sept. 2010.

[56] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2011.

[57] D. Shah and D. Wischik. Principles of resource allocation in networks. In *Proceedings of the ACM SIGCOMM Education Workshop*, May 2011.

[58] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol, Apr. 2003. RFC3530. `http://www.ietf.org/rfc/rfc3530.txt`.

[59] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2012.

[60] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[61] D. Shue and M. J. Freedman. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2014.

[62] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[63] I. Stoica, H. Abdel-Wahab, and A. Pothen. A microeconomic scheduler for parallel computers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1994.

[64] I. E. Sutherland. A futures market in computer time. *Communications of the ACM*, 11(6), June 1968.

[65] A. S. Tanenbaum and D. J. Wetherall. *Computer Networks*. Prentice Hall, 5th edition, Oct. 2010.

[66] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[67] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: Practical power-proportionality for data center storage. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2011.

[68] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[69] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 1994.

[70] E. Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *USENIX ;login:*, 33(5), 2008.

[71] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2010.

[72] H. Wang and P. J. Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2014.

[73] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2012.

[74] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 2004.

Simple Testing Can Prevent Most Critical Failures

An Analysis of Production Failures in Distributed Data-intensive Systems

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, Michael Stumm
University of Toronto

Abstract

Large, production quality distributed systems still fail periodically, and do so sometimes catastrophically, where most or all users experience an outage or data loss. We present the result of a comprehensive study investigating 198 randomly selected, user-reported failures that occurred on Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis, with the goal of understanding how one or multiple faults eventually evolve into a user-visible failure. We found that from a testing point of view, almost all failures require only 3 or fewer nodes to reproduce, which is good news considering that these services typically run on a very large number of nodes. However, multiple inputs are needed to trigger the failures with the order between them being important. Finally, we found the error logs of these systems typically contain sufficient data on both the errors and the input events that triggered the failure, enabling the diagnose and the reproduction of the production failures.

We found the majority of catastrophic failures could easily have been prevented by performing simple testing on error handling code – the last line of defense – even without an understanding of the software design. We extracted three simple rules from the bugs that have lead to some of the catastrophic failures, and developed a static checker, Aspirator, capable of locating these bugs. Over 30% of the catastrophic failures would have been prevented had Aspirator been used and the identified bugs fixed. Running Aspirator on the code of 9 distributed systems located 143 bugs and bad practices that have been fixed or confirmed by the developers.

1 Introduction

Real-world distributed systems inevitably experience outages. For example, an outage to Amazon Web Services in 2011 brought down Reddit, Quora, FourSqure, part of the New York Times website, and about 70 other sites [1], and an outage of Google in 2013 brought down Internet traffic by 40% [21]. In another incident, a DNS error dropped Sweden off the Internet, where every URL in the `.se` domain became unmappable [46].

Given that many of these systems were designed to be highly available, generally developed using good software engineering practices, and intensely tested, this raises the questions of *why these systems still experience failures* and *what can be done to increase their resiliency*. To help answer these questions, we studied 198 randomly sampled, user-reported failures of five data-intensive distributed systems that were designed to tolerate component failures and are widely used in production environments. The specific systems we considered were Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis.

Our goal is to better understand the specific failure manifestation sequences that occurred in these systems in order to identify opportunities for improving their availability and resiliency. Specifically, we want to better understand how one or multiple errors[1] evolve into component failures and how some of them eventually evolve into service-wide catastrophic failures. Individual elements of the failure sequence have previously been studied in isolation, including root causes categorizations [33, 52, 50, 56], different types of causes including misconfiguraitons [43, 66, 49], bugs [12, 41, 42, 51] hardware faults [62], and the failure symptoms [33, 56], and many of these studies have made significant impact in that they led to tools capable of identifying many bugs (e.g., [16, 39]). However, the entire manifestation sequence connecting them is far less well-understood.

For each failure considered, we carefully studied the failure report, the discussion between users and developers, the logs and the code, and we manually reproduced 73 of the failures to better understand the specific manifestations that occurred.

Overall, we found that the error manifestation sequences tend to be relatively complex: more often than not, they require an unusual sequence of multiple events with specific input parameters from a large space to lead the system to a failure. This is perhaps not surprising considering that these systems have undergone thorough testing using unit tests, random error injections [18], and static bug finding tools such as FindBugs [32], and they are deployed widely and in constant use at many organization. But it does suggest that top-down testing, say

[1] Throughout this paper, we use the following standard terminology [36]. A *fault* is the initial root cause, which could be a hardware malfunction, a software *bug*, or a misconfiguration. A fault can produce abnormal behaviors referred to as *errors*, such as system call error return or Java exceptions. Some of the errors will have no user-visible side-effects or may be appropriately handled by software; other errors manifest into a *failure*, where the system malfunction is noticed by end users or operators.

using input and error injection techniques, will be challenged by the large input and state space. This is perhaps why these studied failures escaped the rigorous testing used in these software projects.

We further studied the characteristics of a specific subset of failures — the catastrophic failures that affect all or a majority of users instead of only a subset of users. Catastrophic failures are of particular interest because they are the most costly ones for the vendors, and they are not supposed to occur as these distributed systems are designed to withstand and automatically recover from component failures. Specifically, we found that:

> *almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signaled in software.*

While it is well-known that error handling code is often buggy [24, 44, 55], its sheer prevalence in the causes of the catastrophic failures is still surprising. Even more surprising given that the error handling code is the last line of defense against failures, we further found that:

> *in 58% of the catastrophic failures, the underlying faults could easily have been detected through simple testing of error handling code.*

In fact, in 35% of the catastrophic failures, the faults in the error handling code fall into three trivial patterns: (i) the error handler is simply empty or only contains a log printing statement, (ii) the error handler aborts the cluster on an overly-general exception, and (iii) the error handler contains expressions like "FIXME" or "TODO" in the comments. These faults are easily detectable by tools or code reviews without a deep understanding of the runtime context. In another 23% of the catastrophic failures, the error handling logic of a non-fatal error was so wrong that any statement coverage testing or more careful code reviews by the developers would have caught the bugs.

To measure the applicability of the simple rules we extracted from the bugs that have lead to catastrophic failures, we implemented Aspirator, a simple static checker. Aspirator identified 121 *new* bugs and 379 bad practices in 9 widely used, production quality distributed systems, despite the fact that these systems already use state-of-the-art bug finding tools such as FindBugs [32] and error injection tools [18]. Of these, 143 have been fixed or confirmed by the systems' developers.

Our study also includes a number of additional observations that may be helpful in improving testing and debugging strategies. We found that 74% of the failures are deterministic in that they are guaranteed to manifest with an appropriate input sequence, that almost all failures are guaranteed to manifest on no more than three nodes, and that 77% of the failures can be reproduced by a unit test.

Software	lang.	failures		
		total	sampled	catastrophic
Cassandra	Java	3,923	40	2
HBase	Java	5,804	41	21
HDFS	Java	2,828	41	9
MapReduce	Java	3,469	38	8
Redis	C	1,192	38	8
Total	–	17,216	198	48

Table 1: Number of reported and sampled failures for the systems we studied, and the catastrophic ones from the sample set.

Moreover, in 76% of the failures, the system emits explicit failure messages; and in 84% of the failures, all of the triggering events that caused the failure are printed into the log before failing. All these indicate that the failures can be diagnosed and reproduced in a reasonably straightforward manner, with the primary challenge being to have to sift through relatively noisy logs.

2 Methodology and Limitations

We studied 198 randomly sampled, real world failures reported on five popular distributed data-analytic and storage systems, including HDFS, a distributed file system [27]; Hadoop MapReduce, a distributed data-analytic framework [28]; HBase and Cassandra, two NoSQL distributed databases [2, 3]; and Redis, an in-memory key-value store supporting master/slave replication [54]. We focused on distributed data-intensive systems because they are the building blocks of many internet software services, and we selected the five systems because they are widely used and are considered production quality.

The failures we studied were extracted from the issue tracking databases of these systems. We selected tickets from these databases because of their high quality: each selected failure ticket documents a distinct failure that is confirmed by the developers, the discussions between users and developers, and the failure resolutions in the form of a patch or configuration change. Duplicate failures were marked by the developers, and are excluded from our study.

The specific set of failures we considered were selected from the issue tracking databases as follows. First, we only selected severe failures with the failure ticket *priority field* marked as "Blocker", "Critical", or "Major". Secondly, we only considered tickets dated 2010 or later so as not to include failures of obsolete systems or systems early in their lifetime. Thirdly, we filtered out failures in testing systems by heuristically rejecting failures where the reporter and assignee (i.e., the developer who is assigned to resolve the failure) were the same. Finally, we randomly selected failures from the remaining set to make our observations representative of the entire failure population. Table 1 shows the distribution of the

failure sets considered amongst the five systems and their sampling rates.

For each sampled failure ticket, we carefully studied the failure report, the discussion between users and developers, related error logs, the source code, and patches to understand the root cause and its propagation leading to the failure. We also manually reproduced 73 of the failures to better understand them.

Limitations: as with all characterization studies, there is an inherent risk that our findings may not be representative. In the following we list potential sources of biases and describe how we used our best-efforts to address them.

(1) Representativeness of the selected systems. We only studied distributed, data-intensive software systems. As a result, our findings might not generalize to other types of distributed systems such as telecommunication networks or scientific computing systems. However, we took care to select diverse types of data-intensive programs that include both data-storage and analytical systems, both persistent store and volatile caching, both written in Java and C, both master-slave and peer-to-peer designs. (HBase, HDFS, Hadoop MapReduce, and Redis use master-slave design, while Cassandra uses a peer-to-peer gossiping protocol.) At the very least, these projects are widely used: HDFS and Hadoop MapReduce are the main elements of the Hadoop platform, which is the predominant big-data analytic solution [29]; HBase and Cassandra are the top two most popular wide column store system [30], and Redis is the most popular key-value store system [53].

Our findings also may not generalize to systems earlier in their development cycle since we only studied systems considered production quality. However, while we only considered tickets dated 2010 or later to avoid bugs in premature systems, the buggy code may have been newly added. Studying the evolutions of these systems to establish the correlations between the bug and the code's age remains as the future work.

(2) Representativeness of the selected failures. Another potential source of bias is the specific set of failures we selected. We only studied tickets found in the issue-tracking databases that are intended to document software bugs. Other errors, such as misconfigurations, are more likely to be reported in user discussion forums, which we chose not to study because they are much less rigorously documented, lack authoritative judgements, and are often the results of trivial mistakes. Consequently, we do not draw any conclusions on the distribution of faults, which has been well-studied in complementary studies [50, 52]. Note, however, that it can be hard for a user to correctly identify the nature of the cause of a failure; therefore, our study still includes failures that stem from misconfigurations and hardware faults.

Symptom	all	catastrophic
Unexpected termination	74	17 (23%)
Incorrect result	44	1 (2%)
Data loss or potential data loss*	40	19 (48%)
Hung System	23	9 (39%)
Severe performance degradation	12	2 (17%)
Resource leak/exhaustion	5	0 (0%)
Total	198	48 (24%)

Table 2: Symptoms of failures observed by end-users or operators. The right-most column shows the number of catastrophic failures with "%" identifying the percentage of catastrophic failures over all failures with a given symptom. *: examples of potential data loss include under-replicated data blocks.

In addition, we excluded duplicated bugs from our study so that our study reflects the characteristics of distinct bugs. One could argue that duplicated bugs should not be removed because they happened more often. There were only a total of 10 duplicated bugs that were excluded from our original sample set. Therefore they would not significantly change our conclusions even if they were included.

(3) Size of our sample set. Modern statistics suggests that a random sample set of size 30 or more is large enough to represent the entire population [57]. More rigorously, under standard assumptions, the Central Limit Theorem predicts a 6.9% margin of error at the 95% confidence level for our 198 random samples. Obviously, one can study more samples to further reduce the margin of error.

(4) Possible observer errors. To minimize the possibility of observer errors in the qualitative aspects of our study, all inspectors used the same detailed written classification methodology, and all failures were separately investigated by two inspectors before consensus was reached.

3 General Findings

This section discusses general findings from the entire failure data set in order to provide a better understanding as to how failures manifest themselves. Table 2 categorizes the symptoms of the failures we studied.

Overall, our findings indicate that the failures are relatively complex, but they identify a number of opportunities for improved testing. We also show that the logs produced by these systems are rich with information, making the diagnosis of the failures mostly straightforward. Finally, we show that the failures can be reproduced offline relatively easily, even though they typically occurred on long-running, large production clusters. Specifically, we show that most failures require no more 3 nodes and no more than 3 input events to reproduce, and most failures are deterministic. In fact, most of them can be reproduced with unit tests.

Num. of events	%	
1	23%	single event
2	50%	multiple events: 77%
3	17%	
4	5%	
> 4	5%	

Table 3: Minimum number of input events required to trigger the failures.

Input event type	%
Starting a service	58%
File/database write from client	32%
Unreachable node (network error, crash, etc.)	24%
Configuration change	23%
Adding a node to the running system	15%
File/database read from client	13%
Node restart (intentional)	9%
Data corruption	3%
Other	4%

Table 4: Input events that led to failures. The % column reports the percentage of failure where the input event is required to trigger the failure. Most failures require multiple preceding events, so the sum of the "%" column is greater than 100%.

3.1 Complexity of Failures

Overall, our findings indicate that the manifestations of the failures are relatively complex.

Finding 1 *A majority (77%) of the failures require more than one input event to manifest, but most of the failures (90%) require no more than 3.* (See Table 3.)

Figure 1 provides an example where three input events are required for the failure to manifest.

Table 4 categorizes the input events that lead to failures into 9 categories. We consider these events to be "input events" from a testing and diagnostic point of view — some of the events (e.g., "unreachable node", "data corruption") are not strictly user inputs but can easily be emulated by a tester or testing tools. Note that many of the events have specific requirements for a failure to manifest (e.g., a "file write" event needs to occur on a particular data block), making the input event space to explore for testing immensely large.

Of the 23% of failures that require only a single event to manifest, the event often involves rarely used or newly introduced features, or are caused by concurrency bugs.

Finding 2 *The specific order of events is important in 88% of the failures that require multiple input events.*

Obviously, most of the individual events in Table 4 are heavily exercised and tested (e.g., read and write), which is why only in minority of cases will a single input event induce a failure. In most cases, a specific combination and sequence of multiple events is needed to transition the system into a failed state. Consider the failure example shown in Figure 1. While the events "upload file", "append to file", and "add another datanode" are not problematic individually, the combination of the first two will lead the system into an error state, and the last event actually triggers the failure.

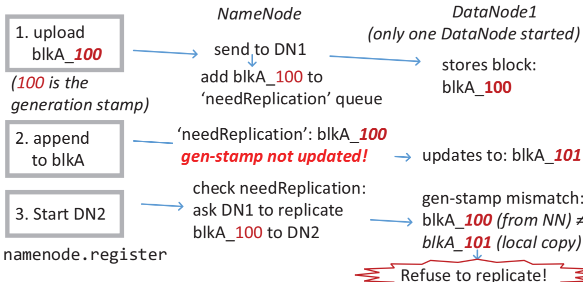


Figure 1: An HDFS failure where a data block remains under-replicated, potentially leading to a data loss. Three input events are needed (shown in boxes): (1) the user uploads a data block, causing HDFS to assign a generation stamp. NameNode (NN) asks DataNode1 (DN1) to store this block, and because this block is currently under-replicated, adds it to `needReplication` queue. (2) the user appends to this block, causing DN1 to increment the generation stamp from 100 to 101. However, the generation stamp in the `needReplication` queue is not updated – an error. (3) DN2 is started, so NN asks DN1 to replicate the block to DN2. But since the generation stamps from `needReplication` queue and DN1 do not match, DN1 keeps refusing to replicate.

Finding 1 and 2 show the complexity of failures in large distributed system. To expose the failures in testing, we need to not only explore the *combination* of multiple input events from an exceedingly large event space, we also need to explore different *permutations*.

3.2 Opportunities for Improved Testing

Additional opportunities to improve existing testing strategies may be found when considering the types of input events required for a failure to manifest. We briefly discuss some of the input event types of Table 4.

Starting up services: More than half of the failures require the start of some services. This suggests that the starting of services — especially more obscure ones — should be more heavily tested. About a quarter of the failures triggered by starting a service occurred on systems that have been running for a long time; e.g., the HBase "Region Split" service is started only when a table grows larger than a threshold. While such a failure may seem hard to test since it requires a long running system, it can be exposed intentionally by forcing a start of the service during testing.

Number of nodes	cumulative distribution function	
	all failures	catastrophic
1	37%	43%
2	84%	86%
3	98%	98%
> 3	100%	100%

Table 5: Min. number of nodes needed to trigger the failures.

Unreachable nodes: 24% of the failures occur because a node is unreachable. This is somewhat surprising given that network errors and individual node crashes are expected to occur regularly in large data centers [14]. This suggests that tools capable of injecting network errors systematically [18, 23, 65] should be used more extensively when inputing other events during testing.

Configuration changes: 23% of the failures are caused by configuration changes. Of those, 30% involve misconfigurations. The remaining majority involve valid changes to enable certain features that may be rarely-used. While the importance of misconfigurations have been observed in previous studies [22, 50, 66], only a few techniques exist to automatically explore configurations changes and test the resulting reaction of the system [19, 40, 63]. This suggests that testing tools should be extended to combine (both valid and invalid) configuration changes with other operations.

Adding a node: 15% of the failures are triggered by adding a node to a running system. Figure 1 provides an example. This is somewhat alarming, given that elastically adding and removing nodes is one of the principle promises of "cloud computing". It suggests that adding nodes needs to be tested under more scenarios.

The production failures we studied typically manifested themselves on configurations with a large number of nodes. This raises the question of how many nodes are required for an effective testing and debugging system.

Finding 3 *Almost all (98%) of the failures are guaranteed to manifest on no more than 3 nodes. 84% will manifest on no more than 2 nodes.* (See Table 5.)

The number is similar for catastrophic failures. Finding 3 implies that it is not necessary to have a large cluster to test for and reproduce failures.

Note that Finding 3 does *not* contradict the conventional wisdom that distributed system failures are more likely to manifest on large clusters. In the end, testing is a probabilistic exercise. A large cluster usually involves more diverse workloads and fault modes, thus increasing the chances for failures to manifest. However, what our finding suggests is that it is not necessary to have a large cluster of machines to expose bugs, as long as the specific sequence of input events occurs.

We only encountered one failure that required a larger number of nodes (over 1024): when the number of simultaneous Redis client connections exceeded the OS limit, `epoll()` returned error, which was not handled properly, causing the entire cluster to hang. All of the other failures require fewer than 10 nodes to manifest.

Software	num. of deterministic failures
Cassandra	76% (31/41)
HBase	71% (29/41)
HDFS	76% (31/41)
MapReduce	63% (24/38)
Redis	79% (30/38)
Total	74% (147/198)

Table 6: Number of failures that are deterministic.

Source of non-determinism	number
Timing btw. input event & internal exe. event	27 (53%)
Multi-thread atomicity violation	13 (25%)
Multi-thread deadlock	3 (6%)
Multi-thread lock contention (performance)	4 (8%)
Other	4 (8%)
Total	51 (100%)

Table 7: Break-down of the non-deterministic failures. The "other" category is caused by nondeterministic behaviors from the OS and third party libraries.

3.3 The Role of Timing

A key question for testing and diagnosis is whether the failures are guaranteed to manifest if the required sequence of input events occur (i.e., *deterministic failures*), or not (i.e., *non-deterministic failures*)?

Finding 4 *74% of the failures are deterministic — they are guaranteed to manifest given the right input event sequences.* (See Table 6.)

This means that for a majority of the failures, we only need to explore the combination and permutation of input events, but no additional timing relationship. This is particularly meaningful for testing those failures that require long-running systems to manifest. As long as we can simulate those events which typically only occur on long running systems (e.g., region split in HBase typically only occurs when the region size grows too large), we can expose these deterministic failures. Moreover, the failures can still be reproduced after inserting additional log output, enabling tracing, or using debuggers.

Finding 5 *Among the 51 non-deterministic failures, 53% have timing constraints only on the input events.* (See Table 7.)

These constraints require an input event to occur either before or after some software internal execution event such as a procedure call. Figure 2 shows an example. In addition to the order of the four input events (that can be

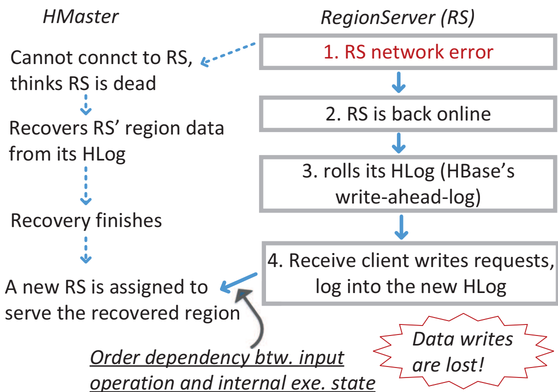


Figure 2: A non-deterministic failure in HBase with timing requirements (shown with solid arrows) only on input events (boxed). Some newly written data will be lost because when HMaster assigns a new region server, it only recovered the old HLog that does not contain the newly written data.

```
write_lock();
/* remove a
 * large directory */
write_unlock();
```

Critical region is too large, causing concurrent write requests to hang

Figure 3: Performance degradation in HDFS caused by a single request to remove a large directory.

controlled by a tester), the additional requirement is that the client write operations must occur before HMaster assigns the region to a new Region Server, which cannot be completely controlled by the user.

Note that these non-deterministic dependencies are still easier to test and debug than non-determinisms stemming from multi-threaded interleavings, since at least one part of the timing dependency can be controlled by testers. Testers can carefully control the timing of the input events to induce the failure. Unit tests and model checking tools can further completely manipulate such timing dependencies by controlling the timing of both the input events and the call of internal procedures. For example, as part of the patch to fix the bug in Figure 2, developers used a unit test that simulated the user inputs *and* the dependencies with HMaster's operations to deterministically reproduce the failure.

The majority of the remaining 24 non-deterministic failures stem from shared-memory multi-threaded interleavings. We observed three categories of concurrency bugs in our dataset: atomicity violation [42], deadlock, and lock contention that results in performance degradation. It is much harder to expose and reproduce such failures because it is hard for users or tools to control timing, and adding a single logging statement can cause the failure to no longer expose itself. We reproduced 10 of these non-deterministic failures and found the atomicity violations and deadlocks the most difficult to reproduce (we had to manually introduce additional timing delays,

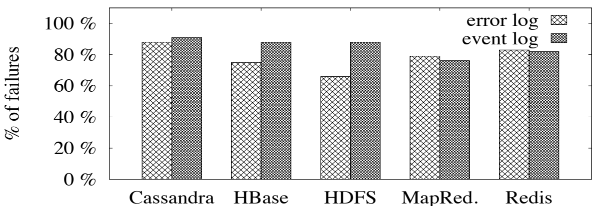


Figure 4: The logging comprehensiveness of the studied failures. Logging of both the input events and errors are considered. For failures requiring multiple events to trigger, we count it as "logged" only when *all* the required events are logged.

like `Thread.sleep()` in the code to trigger the bugs). The lock contention cases, however, are not as difficult to reproduce. Figure 3 shows an example where a bug caused unnecessary lock contention.

3.4 Logs Enable Diagnosis Opportunities

Overall, we found the logs output by the systems we studied to be rich with useful information.We assume the default logging verbosity level is used.

Finding 6 *76% of the failures print explicit failure-related error messages.* (See Figure 4.)

This finding somewhat contradicts the findings of our previous study [67] on failures in non-distributed systems, including Apache httpd, PostgreSQL, SVN, squid, and GNU Coreutils, where only 43% of failures had explicit failure-related error messages logged. We surmise there are three possible reasons why developers output log messages more extensively for the distributed systems we studied. First, since distributed systems are more complex, and harder to debug, developers likely pay more attention to logging. Second, the horizontal scalability of these systems makes the performance overhead of outputing log message less critical. Third, communicating through message-passing provides natural points to log messages; for example, if two nodes cannot communicate with each other because of a network problem, both have the opportunity to log the error.

Finding 7 *For a majority (84%) of the failures, all of their triggering events are logged.* (See Figure 4.)

This suggests that it is possible to deterministically replay the majority of failures *based on the existing log messages alone*. Deterministic replay has been widely explored by the research community [4, 13, 15, 26, 35, 47, 61]. However, these approaches are based on intrusive tracing with significant runtime overhead and the need to modify software/hardware.

Finding 8 *Logs are noisy: the median of the number of log messages printed by each failure is 824.*

Software	% of failures reproducible by unit test
Cassandra	73% (29/40)
HBase	85% (35/41)
HDFS	82% (34/41)
MapReduce	87% (33/38)
Redis	58% (22/38)
Total	77% (153/198)

Table 8: Percentage of failures that can be reproduced by a unit test. The reason that only a relatively small number of Redis failures can be reproduced by unit tests is that its unit-test framework is not as powerful, being limited to command-line commands. Consequently, it cannot simulate many errors such as node failure, nor can it call some internal functions directly.

This number was obtained when reproducing 73 of the 198 failures with a *minimal* configuration and using a minimal workload that is just sufficient to reproduce the failure. Moreover, we did not count the messages printed during the start-up and shut-down phases.

This suggests that manual examination of the log files could be tedious. If a user only cares about the error symptoms, a selective `grep` on the error verbosity levels will reduce noise since a vast majority of the printed log messages are at INFO level. However, the input events that triggered the failure are often logged at INFO level. Therefore to further infer the input events one has to examine almost every log message. It would be helpful if existing log analysis techniques [5, 6, 48, 64] and tools were extended so they can infer the relevant error and input event messages by filtering out the irrelevant ones.

3.5 Failure Reproducibility

Conventional wisdom has it that failures which occur on large, distributed system in production are extremely hard to reproduce off-line. The users' input may be unavailable due to privacy concerns, the difficulty in setting up an environment that mirrors the one in production, and the cost of third-party libraries, are often reasons cited as to why it is difficult for vendors to reproduce production failures. Our finding below indicates that failure reproduction might not be as hard as it is thought to be.

Finding 9 *A majority of the production failures (77%) can be reproduced by a unit test.* (See Table 8.)

While this finding might sound counter-intuitive, it is not surprising given our previous findings because: (1) in Finding 4 we show that 74% of the failures are deterministic, which means the failures can be reproduced with the same operation sequence; and (2) among the remaining non-deterministic failures, in 53% of the cases the timing can be controlled through unit tests.

Specific data values are not typically required to reproduce the failures; in fact, *none* of the studied failures required specific values of user's data contents. Instead, only the required input sequences (e.g., file write, disconnect a node, etc.) are needed.

```
public void testLogRollAfterSplitStart {
  startMiniCluster(3);
   // create an HBase cluster with 1 master and 2 RS

  HMaster.splitHLog();
   // simulate a hlog splitting (HMaster's recovery
   // of RS' region data) when RS cannot be reached

  RS.rollHLog();
   // simulate the region server's log rolling event

  for (i = 0; i < NUM_WRITES; i++)
    writer.append(..); // write to RS' region

  HMaster.assignNewRS();
   // HMaster assigns the region to a new RS

  assertEquals (NUM_WRITES, countWritesHLog());
   // Check if any writes are lost
}
```

Figure 6: Unit test for the failure shown in Figure 2.

Figure 6 shows how a unit test can simulate the non-deterministic failure of Figure 2. It simulates a mini-cluster by starting three processes running as three nodes. It further simulates the key input events, including HMaster's log split, Region Server's log rolling, and the write requests. The required dependency where the client must send write requests before the master re-assigns the recovered region is also controlled by this unit test.

The failures that cannot be reproduced easily either depend on a particular execution environment (such as OS version or third party libraries), or were caused by non-deterministic thread interleavings.

4 Catastrophic Failures

Table 2 in Section 3 shows that 48 failures in our entire failure set have catastrophic consequences. We classify a failure to be catastrophic when it prevents *all or a majority of the users* from their normal access to the system. In practice, these failures result in cluster-wide outage, a hung cluster, or a loss to all or a majority of the user data. Note that a bug resulting in under-replicated data blocks is *not* considered as catastrophic, even when it affect all data blocks, because it does not prevent users from their normal read and write to their data yet. We specifically study the catastrophic failures because they are the ones with the largest business impact to the vendors.

The fact that there are so many catastrophic failures is perhaps surprising given that the systems considered all have High Availability (HA) mechanisms designed to prevent component failures from taking down the entire service. For example, all of the four systems with a master-slave design — namely HBase, HDFS, MapReduce, and Redis — are designed to, on a master node failure, automatically elect a new master node and fail

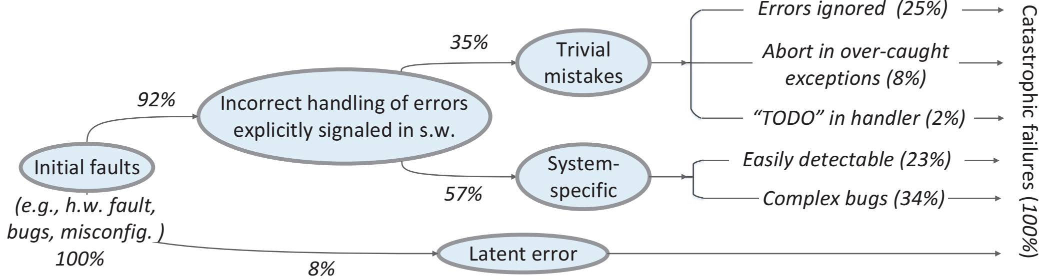


Figure 5: Break-down of all catastrophic failures by their error handling.

over to it.[2] Cassandra is a peer-to-peer system, thus by design it avoids single points of failure. Then why do catastrophic failures still occur?

Finding 10 *Almost all catastrophic failures (92%) are the result of incorrect handling of non-fatal errors explicitly signaled in software.* (See Figure 5.)

These catastrophic failures are the result of more than one fault triggering, where the initial fault, whether due to a hardware fault, a misconfiguration, or a bug, first manifests itself explicitly as a non-fatal error — for example by throwing an exception or having a system call return an error. This error need not be catastrophic; however in the vast majority of cases, the handling of the explicit error was faulty, resulting in an error manifesting itself as a catastrophic failure.

This prevalence of incorrect error handling is unique to catastrophic failures. In comparison, only 25% of the non-catastrophic failures in our study involve incorrect error handling, indicating that in non-catastrophic failures, error handling was mostly effective in preventing the errors from taking down the entire service.

Overall, we found that the developers are good at anticipating possible errors. In all but one case, the errors were checked by the developers. The only case where developers did not check the error was an unchecked error system call return in Redis. This is different from the characteristics observed in previous studies on file system bugs [24, 41, 55], where many errors weren't even checked. This difference is likely because (i) the Java compiler forces developers to catch all the checked exceptions; and (ii) a variety of errors are expected to occur in large distributed systems, and the developers program more defensively. However, we found they were often simply sloppy in handling these errors. This is further corroborated in Findings 11 and 12 below. To be fair, we should point out that our findings are skewed in the sense that our study did not expose the many errors that are correctly caught and handled.

[2] We assume the HA feature is always enabled when classifying catastrophic failures. We did not classify failures as catastrophic if HA was not enabled and the master node failed, even though it would likely have affected all the users of the system. This is because such failures are not unique compared to the other failures we studied — they just happened to have occurred on the master node.

Nevertheless, the correctness of error handling code is particularly important given their impact. Previous studies [50, 52] show that the initial faults in distributed system failures are highly diversified (e.g., bugs, misconfigurations, node crashes, hardware faults), and in practice it is simply impossible to eliminate them all in large data centers [14]. It is therefore unavoidable that some of these faults will manifest themselves into errors, and error handling then becomes the last line of defense [45].

Of the catastrophic failures we studied, only four were not triggered by incorrect error handling. Three of them were because the servers mistakenly threw fatal exceptions that terminated all the clients, i.e., the clients' error handling was correct. The other one was a massive performance degradation when a bug disabled DNS look-up result caching.

4.1 Trivial Mistakes in Error Handlers

Finding 11 *35% of the catastrophic failures are caused by trivial mistakes in error handling logic — ones that simply violate best programming practices; and that can be detected without system specific knowledge.*

Figure 5 further breaks down the mistakes into three categories: (i) the error handler ignores explicit errors; (ii) the error handler over-catches an exception and aborts the system; and (iii) the error handler contains "TODO" or "FIXME" in the comment.

25% of the catastrophic failures were caused by ignoring explicit errors (an error handler that only logs the error is also considered as ignoring the error). For systems written in Java, the exceptions were all explicitly thrown, whereas in Redis they were system call error returns. Figure 7 shows a data loss in HBase caused by ignoring an exception. Ignoring errors and allowing them to propagate is known to be bad programming practice [7, 60], yet we observed this lead to many catastrophic failures. At least the developers were careful at logging the errors: all the errors were logged except for one case where the Redis developers did not log the error system call return.

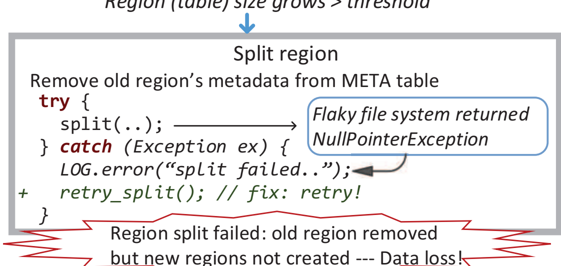


Figure 7: A data loss in HBase where the error handling was simply empty except for a logging statement. The fix was to retry in the exception handler.

```
    try {
      namenode.registerDatanode();
+   } catch (RemoteException e) {
+     // retry.
    } catch (Throwable t) {
      System.exit(-1);
    }
```

RemoteExcepion is thrown due to glitch in namenode

Only intended for IncorrectVersionException

Figure 8: Entire HDFS cluster brought down by an over-catch.

User: MapReduce jobs hang when a rare Resource Manager restart occurs. *I have to ssh to every one of our 4000 nodes in a cluster and try to kill all the running Application Manager.*

Patch:

```
 catch (IOException e) {
-  // TODO
   LOG("Error event from RM: shutting down..");
+  // This can happen if RM has been restarted. Must clean up.
+  eventHandler.handle(..);
 }
```

Figure 9: A catastrophic failure in MapReduce where developers left a "TODO" in the error handler.

Another 8% of the catastrophic failures were caused by developers prematurely aborting the entire cluster on a non-fatal exception. While in principle one would need system specific knowledge to determine when to bring down the entire cluster, the aborts we observed were all within *exception over-catch*, where a higher level exception is used to catch multiple different lower-level exceptions. Figure 8 shows such an example. The `exit()` was intended only for IncorrectVersionException. However, the developers catch a high-level exception: Throwable. Consequently, when a glitch in the namenode caused `registerDatanode()` to throw RemoteException, it was over-caught by Throwable and thus brought down every datanode. The fix was to handle RemoteException explicitly, so that only IncorrectVersionException would fall through. However, this is still bad practice since later when the code evolves, some other exceptions may be over-caught again. The safe practice is to catch the precise exception [7].

Figure 9 shows an even more obvious mistake, where the developers only left a comment "TODO" in the handler logic in addition to a logging statement. While this error would only occur rarely, it took down a production cluster of 4,000 nodes.

1: HMaster assigns region R from RS1 to RS2

2: RS2 opens R

4: RS2 dies

5: HMaster assigns R to RS3

6: RS3 opens R

```
HMaster.OpenedRegionHandler()
 try {
 3: newlyOpened(getZookeeper(), R);
   // inform zookeeper
   // that R is OPENED
 } catch (KeeperException e) {
   // R is already OPENED...
 7:abort("Error in .. ..");
 }
```

Figure 10: A catastrophic failure where the error handling code was wrong and simply not tested at all. A rare sequence of events caused `newlyOpened()` to throw a rare KeeperException, which simply took down the entire HBase cluster.

4.2 System-specific Bugs

The other 57% of the catastrophic failures are caused by incorrect error handling where system-specific knowledge is required to detect the bugs. (See Figure 5.)

Finding 12 *In 23% of the catastrophic failures, while the mistakes in error handling were system specific, they are still easily detectable. More formally, the incorrect error handling in these cases would be exposed by 100% statement coverage testing on the error handling logic.*

In other words, once the problematic basic block in the error handling code is triggered, the failure is guaranteed to be exposed. This suggests that these basic blocks were completely faulty and simply never properly tested. Figure 10 shows such an example. Once a test case can deterministically trigger KeeperException, the catastrophic failure will be triggered with 100% certainty.

Hence, a good strategy to prevent these failures is to start from existing error handling logic and try to reverse engineer test cases that trigger them. For example, symbolic execution techniques [8, 10] could be extended to purposefully reconstruct an execution path that can reach the error handling code block, instead of blindly exploring every execution path from the system entry points.

While high statement coverage on error handling code might seem difficult to achieve, aiming for higher statement coverage in testing might still be a better strategy than a strategy of applying random fault injections. For example, the failure in Figure 10 requires a very rare combination of events to trigger the buggy error handler. Our finding suggests that a "bottom-up" approach could be more effective: start from the error handling logic and reverse engineer a test case to expose errors there.

Existing testing techniques for error handling logic primarily use a "top-down" approach: start the system using testing inputs or model-checking [23, 65], and actively inject errors at different stages [9, 18, 44]. Tools like LFI [44] and Fate&Destini [23] are intelligent to inject errors only at appropriate points and avoid duplicated injections. Such techniques inevitably have greatly

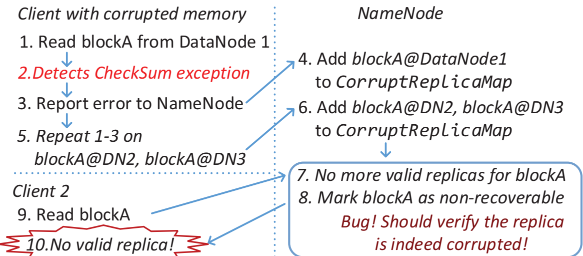


Figure 11: A massive data loss for all clients in HDFS. A client with corrupted RAM reported data corruption on almost every block it reads to the namenode. Instead of verifying the checksum on datanodes, namenode blindly trusts the faulty client and marks the blocks as permanently corrupted, causing a massive data loss to all clients.

improved the reliability of software systems. In fact, Hadoop developers have their own error injection framework to test their systems [18], and the production failures we studied are likely the ones missed by such tools.

However, our findings suggest that it could be challenging for such "top-down" approaches to further expose these remaining production failures. They require rare sequence of input events to first take the system to a rare state, before the injected error can take down the service. In addition, 38% of the failures only occur in long-running systems. Therfore, the possible space of input events would simply be untractable.

Complex bugs: the remaining 34% of catastrophic failures involve complex bugs in the error handling logic. These are the cases where developers did not anticipate certain error scenarios. As an example, consider the failure shown in Figure 11. While the handling logic makes sense for a majority of the checksum errors, it did not consider the scenario where a single client reports a massive number of corruptions (due to corrupt RAM) in a very short amount of time. These type of errors — which are almost byzantine — are indeed the hardest to test for. Detecting them require both understanding how the system works and anticipating all possible real-world failure modes. While our study cannot provide constructive suggestions on how to identify such bugs, we found they only account for one third of the catastrophic failures.

4.3 Discussion

While we show that almost all of the catastrophic failures are the result of incorrect error handling, it could be argued that most of the code is reachable from error handling blocks in real-world distributed system, therefore most of the bugs are "incorrect error handling". However, our findings suggest many of the bugs can be detected by only examining the exception handler blocks (e.g., the `catch` block in Java). As we show will in Table 9, the number of `catch` blocks in these systems is relatively small. For example, in HDFS, there are only 2652 `catch` blocks. In particular, the bugs belonging to the "trivial mistakes" category in Finding 11 can be easily detected by only examining these `catch` blocks.

An interesting question is whether the outages from large internet software vendors are also the result of incorrect error handling. While we cannot answer this rigorously without access to their internal failure databases, the postmortem analysis of some of the most visible outages are released to the public. Interestingly, some of the anecdotal outages are the result of incorrect error handling. For example, in an outage that brought down facebook.com for approximately 2.5 hours, which at that time was "the worst outage Facebook have had in over four years", "the key flaw that caused the outage to be so severe was an unfortunate handling of an error condition" [17]. In the outage of Amazon Web Services in 2011 [59] that brought down Reddit, Quora, FourSquare, parts of the New York Times website, and about 70 other sites, the initial cause was a configuration change that mistakenly routed production traffic to a secondary network that was not intended for a heavy workload. Consequently, nodes start to fail. What lead this to further propagate into a service-level failure was the incorrect handling of node-failures — "the nodes failing to find new nodes did not back off aggressively enough when they could not find space, but instead, continued to search repeatedly". This caused even more network traffic, and eventually lead to the service-level failure.

5 Aspirator: A Simple Checker

In Section 4.1, we observed that some of the most catastrophic failures are caused by trivial mistakes that fall into three simple categories: (i) error handler is empty; (ii) error handler over-catches exceptions and aborts; and (iii) error handler contains phrases like "TODO" and "FIXME". To measure the applicability of these simple rules, we built a rule-based static checker, Aspirator, capable of locating these bug patterns. Next we discuss how Aspirator is implemented and the results of applying it to a number of systems.

5.1 Implementation of Aspirator

We implemented Aspirator using the Chord static analysis framework [11] on Java bytecode. Aspirator works as follows: it scans Java bytecode, instruction by instruction. If an instruction can throw exception e, Aspirator identifies and records the corresponding `catch` block for e. Aspirator emits a warning if the `catch` block is empty or just contains a log printing statement, or if the `catch` block contains "TODO" or "FIXME" com-

ments in the corresponding source code. It also emits a warning if a `catch` block for a higher-level exception (e.g., Exception or Throwable) might catch *multiple* lower-level exceptions *and* at the same time calls `abort` or `System.exit()`. Aspirator is capable of identifying these *over-catches* because when it reaches a `catch` block, it knows exactly which exceptions from which instructions the `catch` block handles.

Not every empty `catch` block is necessarily a bad practice or bug. Consider the following example where the exception is handled outside of the `catch` block:

```
uri = null;
try {
  uri = Util.fileAsURI(new File(uri));
} catch (IOException ex) { /* empty */ }
if (uri == null) { // handle it here!
```

Therefore Aspirator will not emit a warning on an empty `catch` block if both of the following conditions are true: (i) the corresponding `try` block modifies a variable V; and (ii) the value of V is checked in the basic block following the `catch` block. In addition, if the last instruction in the corresponding `try` block is a `return`, `break`, or `continue`, and the block after the `catch` block is not empty, Aspirator will not report a warning if the `catch` block is empty because all the logic after the `catch` block is in effect exception handling.

Aspirator further provides runtime configuration options to allow programmers to adjust the trade-offs between false positives and false negatives. It allows programmers to specify exceptions that should not result in a warning. In our testing, we ignored all instances of the FileNotFound exception, because we found the vast majority of them do not indicate a true error. Aspirator also allows programmers to exclude certain methods from the analysis. In our testing, we use this to suppress warnings if the ignored exceptions are from a `shutdown`, `close` or `cleanup` method — exceptions during a cleanup phase are likely less important because the system is being brought down anyway. Using these two heuristics did not affect Aspirator's capability to detect the trivial mistakes leading to catastrophic failures in our study, yet significantly reduce the number of false positives.

Limitations: As a proof-of-concept, Aspirator currently only works on Java and other languages that are compatible with Java bytecode (e.g., Scala), where exceptions are supported by the language and are required to be explicitly caught. The main challenge to extend Aspirator to non-Java programs is to identify the error conditions. However, some likely error conditions can still be easily identified, including system call error returns, switch fall-through, and calls to `abort()`.

In addition, Aspirator cannot estimate the criticality of the warnings it emits. Hence, not every warning emitted will identify a bug that could lead to a failure; in fact, some false positives are emitted. However, because Aspirator provides, with each warning, a list of caught exceptions together with the instructions that throw them, developers in most cases will be able to quickly assess the criticality of each warning and possibly annotate the program to suppress specific future warnings.

(a)

```
try {
  tableLock.release();
} catch (IOException e) {
  LOG("Can't release lock",
    e);
}
```

hang: lock is never released!

(b)

```
try {
  journal.recoverSegments();
} catch (IOException ex) {
}
```

Cannot apply the updates from Edit log, ignoring it can cause dataloss!

Figure 12: Two new bugs found by Aspirator.

Finally, the functionality of Aspirator could (and probably should) be added to existing static analysis tools, such as FindBugs [32].

5.2 Checking Real-world Systems

We first evaluated Aspirator on the set of catastrophic failures used in our study. If Aspirator had been used and the captured bugs fixed, 33% of the Cassandra, HBase, HDFS, and MapReduce's catastrophic failures we studied could have been prevented.

We then used Aspirator to check the latest stable versions of 9 distributed systems or components used to build distributed systems (e.g., Tomcat web-server). Aspirator's analysis finishes within 15 seconds for each system on a MacBook Pro laptop with 2.7GHz Intel Core i7 processor, and has memory footprints of less than 1.2GB.

We categorize each warning generated by Aspirator into one of three categories: bug, bad practice, and false positive. For each warning, we use our best-effort to understand the consequences of the exception handling logic. Warnings are categorized as bugs only if we could definitively conclude that, once the exception occurs, the handling logic could lead to a failure. They were categorized as false positives if we clearly understood they would not lead to a failure. All other cases are those that we could not definitively infer the consequences of the exception handling logic without domain knowledge. Therefore we conservatively categorize them as bad practices.

Table 9 shows the results. Overall, Aspirator detected 500 new bugs and bad practices along with 115 false positives. Note that most of these systems already run state-of-the-art static checkers like FindBugs [32], which checks for over 400 rules, on every code check-in. Yet Aspirator has found new bugs in all of them.

Bugs: many bugs detected by Aspirator could indeed lead to catastrophic failures. For example, all 4 bugs caught by the abort-in-over-catch checker could bring

System	Handler blocks	Bug					Bad practice					False pos.
		total / confirmed		ignore / abort / todo			total / confirmed		ignore / abort / todo			
Cassandra	4,365	2	2	2	-	-	2	2	2	-	-	9
Cloudstack	6,786	27	24	25	-	-	185	21	182	1	2	20
HDFS	2,652	24	9	23	-	1	32	5	32	-	-	16
HBase	4,995	16	16	11	3	2	43	6	35	5	3	20
Hive	9,948	25	15	23	-	2	54	14	52	-	2	8
Tomcat	5,257	7	4	6	1	-	23	3	17	4	2	30
Spark	396	2	2	-	-	2	1	1	1	-	-	2
YARN/MR2	1,069	13	8	6	-	7	15	3	10	4	1	1
Zookeeper	1,277	5	5	5	-	-	24	3	23	-	1	9
Total	36,745	**121**	**85**	101	4	16	**379**	**58**	354	14	11	115

Table 9: Results of applying Aspirator to 9 distributed systems. If a case belongs to multiple categories (e.g., an empty handler may also contain a "TODO" comment), we count it only once as an ignored exception. The "Handler blocks" column shows the number of exception handling blocks that Aspirator discovered and analyzed. "-" indicates Aspirator reported 0 warning.

down the cluster on an unexpected exception in a similar fashion as in Figure 8. All 4 of them have been fixed.

Some bugs can also cause the cluster to hang. Aspirator detected 5 bugs in HBase and Hive that have a pattern similar to the one depicted in Figure 12 (a). In this example, when `tableLock` cannot be released, HBase only outputs an error message and continues executing, which can deadlock all servers accessing the table. The developers fixed this bug by immediately cleaning up the states and aborting the problematic server [31].

Figure 12 (b) shows a bug that could lead to data loss. An IOException could be thrown when HDFS is recovering user data by replaying the updates from the Edit log. Ignoring it could cause a silent data loss.

Bad practices: the bad practice cases include potential bugs for which we could not definitively determine their consequences without domain expertise. For example, if deleting a temporary file throws an exception and is subsequently ignored, it may be inconsequential. However, it is nevertheless considered a bad practice because it may indicate a more serious problem in the file system.

Some of these cases could as well be false positives. While we cannot determine how many of them are false positives, we did report 87 of the cases that we initially classified as "bad practices" to developers. Among them, 58 were confirmed or fixed, but 17 were rejected. The 17 rejected ones were subsequently classified as "false positives" in Table 9.

False positives: 19% of the warnings reported by Aspirator are false positives. Most of them are due to that Aspirator does not perform inter-procedural analysis. Consider the following example, where an exception is handled by testing the return value of a method call:

```
try {
   set_A();
} catch (SomeException e) { /* empty */ }
if (A_is_not_set()) {/* handle it here! */}
```

In addition to FileNotFound and exceptions from from `shutdown`, `close`, and `cleanup`, Aspirator should have been further configured to exclude the warnings on other exceptions. For example, many of the false positives are caused by empty handlers of Java's reflection related exceptions, such as NoSuchFieldException. Once programmers realize an exception should have been excluded from Aspirator's analysis, they can simply add this exception to Aspirator's configuration file.

5.3 Experience

Interaction with developers: We reported 171 bugs and bad practices to the developers through the official bug tracking website. To this date, 143 have already been confirmed or fixed by the developers (73 of them have been fixed, and the other 70 have been confirmed but not fixed yet), 17 were rejected, and the others have not received any responses.

We received mixed feedback from developers. On the one hand, we received some positive comments like: "*I really want to fix issues in this line, because I really want us to use exceptions properly and never ignore them*", "*No one would have looked at this hidden feature; ignoring exceptions is bad precisely for this reason*", and "*catching Throwable [i.e., exception over-catch] is bad, we should fix these*". On the other hand, we received negative comments like: "*I fail to see the reason to handle every exception*".

There are a few reasons for developers' obliviousness to the handling of errors. First, these ignored errors may not be regarded as critical enough to be handled properly. Often, it is only until the system suffers serious failures will the importance of the error handling be realized by developers. We hope to raise developers' awareness by showing that many of the most catastrophic failures today are caused precisely by such obliviousness to the correctness of error handling logic.

Secondly, the developers may believe the errors would never (or only very rarely) occur. Consider the following code snippet detected by Aspirator from HBase:

```
try {
  t = new TimeRange(timestamp, timestamp+1);
} catch (IOException e) {
  // Will never happen
}
```

In this case, the developers thought the constructor could never throw an exception, so they ignored it (as per the comment in the code). We observed many empty error handlers contained similar comments in multiple systems we checked. We argue that errors that "can never happen" should be handled defensively to prevent them from propagating. This is because developers' judgement could be wrong, later code evolutions may enable the error, and allowing such unexpected errors to propagate can be deadly. In the HBase example above, developers' judgement was indeed wrong. The constructor is implemented as follows:

```
public TimeRange (long min, long max)
throws IOException {
  if (max < min)
    throw new IOException("max < min");
}
```

It could have thrown an IOException when there is an integer overflow, and swallowing this exception could have lead to a data loss. The developers later fixed this by handling the IOException properly.

Thirdly, proper handling of the errors can be difficult. It is often much harder to reason about the correctness of a system's abnormal execution path than its normal execution path. The problem is further exacerbated by the reality that many of the exceptions are thrown by third party components lacking of proper documentations. We surmise that in many cases, even the developers may not fully understand the possible causes or the potential consequences of an exception. This is evidenced by the following code snippet from CloudStack:

```
} catch (NoTransitionException ne) {
  /* Why this can happen? Ask God not me. */
}
```

We observed similar comments from empty exception handlers in other systems as well.

Finally, in reality feature development is often prioritized over exception handling when release deadlines loom. We embarrassingly experienced this ourselves when we ran Aspirator on Aspirator's code: we found 5 empty exception handlers, all of them for the purpose of catching exceptions thrown by the underlying libraries and put there only so that the code would compile.

Good practice in Cassandra: among the 9 systems we checked, Cassandra has the lowest bug-to-handler-block ratio, indicating that Cassandra developers are careful in following good programming practices in exception handling. In particular, the vast majority of the exceptions are handled by recursively propagating them to the callers, and are handled by top level methods in the call graphs. Interestingly, among the 5 systems we studied, Cassandra also has the lowest rate of catastrophic failures in its randomly sampled failure set (see Table 1).

6 Related Work

A number of studies have characterized failures in distributed systems, which led to a much deeper understanding of these failures and hence improved reliability. Our study is the first (to the best of our knowledge) analysis to understand the end-to-end manifestation sequence of these failures. The manual analysis allowed us to find the weakest link on the manifestation sequence for the most catastrophic failures, namely the incorrect error handling. While it is well-known that error handling is a source of many errors, we found that these bugs in error handling code, many of them extremely simple, are the dominant cause of today's catastrophic failures.

Next, we discuss three categories of related work: characterization studies, studies on error handling code, and distributed system testing.

Failure characterization studies Oppenheimer *et al.* eleven years ago studied over 100 failure reports from deployed internet services [50]. They discussed the root causes, time-to-repair, and mitigation strategies of these failures, and summarized a series of interesting findings (e.g., operator mistakes being the most dominant cause). Our study is largely complementary since the open-source projects allow us to examine a richer source of data, including source code, logs, developers' discussions, etc., which were not available for their study. Indeed, as acknowledged by the authors, they "could have been able to learn more about the detailed causes if [they] had been able to examine the system logs and bug tracking database".

Rabkin and Katz [52] analyzed reports from Cloudera's production hadoop clusters. Their study focused on categorizing the root causes of the failures.

Li *et al.* [38] studied bugs in Microsoft Bing's data analytic jobs written in SCOPE. They found that most of the bugs were in the data processing logic and were often caused by frequent change of table schema.

Others studied bugs in non-distributed systems. In 1985, Gray examined over 100 failures from the Tandem [22] operating system, and found operator mistakes and software bugs to be the two major causes. Chou *et al.* [12] studied OS bugs and observed that device drivers are the most buggy. This finding led to many systems and tools to improve device driver quality, and a study [51] ten years later suggested that the quality of device drivers have indeed greatly improved. Lu *et al.* [42] studied concurrency bugs in server programs, and found many inter-

esting findings, e.g., almost all of the concurrency bugs can be triggered using 2 threads.

Study on error handling code Many studies have shown that error handling code is often buggy [24, 44, 55, 58]. Using a static checker, Gunawi *et al.* found that file systems and storage device drivers often do not correctly propagate error code [24]. Fu and Ryder also observed that a significant number of `catch` blocks were empty in many Java programs [20]. But they did not study whether they have caused failures. In a study on field failures with IBM's MVS operating system between 1986 and 1989, Sullivan *et al.* found that incorrect error recovery was the cause of 21% of the failures and 36% of the failures with high impact [58]. In comparison, we find that in the distributed systems we studied, incorrect error handling resulted in 25% of the non-catastrophic failures, and 92% of the catastrophic ones.

Many testing tools can effectively expose incorrect error handling through error injections [18, 23, 44]. Fate&Destini [23] can intelligently inject unique combinations of multiple errors; LFI [44] selectively injects errors at the program/library boundary and avoids duplicated error injections. While these tools can be effective in exposing many incorrect error handling bugs, they all use a "top-down" approach and rely on users/testers to provide workloads to drive the system. In our study, we found that a combination of input events is needed to drive the system to the error state which is hard to trigger using a top-down approach. Our findings suggests that a "bottom-up" approach, which reconstruct test cases from the error handling logic, can effectively expose most faults that lead to catastrophic failures.

Other tools are capable of identify bugs in error handling code via static analysis [24, 55, 67]. EIO [24] uses static analysis to detect error code that is either unchecked or not further propagated. Errlog [67] reports error handling code that is not *logged*. In comparison, our simple checker is complementary. It detects exceptions that are checked but incorrectly handled, regardless whether they are logged or not.

Distributed system testing Model checking [25, 34, 37, 65] tools can be used to systematically explore a large combination of different events. For example, SAMC [37] can intelligently inject multiple errors to drive the target system into a corner case. Our study further helps users make informed decisions when using these tools (e.g., users need to check no more than three nodes).

7 Conclusions

This paper presented an in-depth analysis of 198 user-reported failures in five widely used, data-intensive distributed systems in the form of 12 findings. We found that the error manifestation sequences leading to the failures to be relatively complex. However, we also found that for the most catastrophic failures, almost all of them are caused by incorrect error handling, and 58% of them are trivial mistakes or can be exposed by statement coverage testing.

It is doubtful that existing testing techniques will be successful uncovering many of these error handling bugs. They all use a "top-down" approach: start the system using generic inputs or model-checking [65, 23], and actively inject errors at different stages [9, 18, 44]. However the size of the input and state space, and the fact that a significant number of failures only occur on long-running systems, makes the problem of exposing these bugs intractable. For example, Hadoop has its own error injection framework to test their system [18], but the production failures we studied are likely the ones missed by such tools.

Instead, we suggest a three pronged approach to expose these bugs: (1) use a tool similar to the Aspirator that is capable of identifying a number of trivial bugs; (2) enforce code reviews on error-handling code, since the error handling logic is often simply wrong; and (3) use, for example, extended symbolic execution techniques [8, 10] to purposefully reconstruct execution paths that can reach each error handling code block. Our detailed analysis of the failures and the source code of Aspirator are publicly available at: `http://www.eecg.toronto.edu/failureAnalysis/`.

Acknowledgements

We greatly appreciate the anonymous reviewers, our shepherd Jason Flinn, and Leonid Ryzhyk for their insightful feedback. This research is supported by NSERC Discovery grant, NetApp Faculty Fellowship, and Connaught New Researcher Award.

References

[1] Why Amazon's cloud titanic went down. `http://money.cnn.com/2011/04/22/technology/amazon_ec2_cloud_outage/index.htm`.

[2] Apache Cassandra. `http://cassandra.apache.org`.

[3] Apache HBase. `http://hbase.apache.org`.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[5] I. Beschastnikh, Y. Brun, , J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *Proceedings*

of The International Conference on Software Engineering, ICSE'13, 2013.

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. Mining temporal invariants from partially ordered logs. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML'11, pages 3:1–3:10, 2011.

[7] J. Bloch. *Effective Java (2nd Edition)*. Prentice Hall, 2008.

[8] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[9] Chaos monkey. https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, Feb. 2012.

[11] Chord: A program analysis platform for Java. http://pag.gatech.edu/chord.

[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, 2001.

[13] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405, 2013.

[14] J. Dean. Underneath the covers at Google: current systems and future directions. In *Google I/O*, 2008.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, OSDI'02, 2002.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design and Implementation*, OSDI'00, pages 1–16, 2000.

[17] Facebook: More details on today's outage. https://www.facebook.com/note.php?note_id=431441338919&id=9445547199&ref=mf.

[18] Hadoop team. Fault injection framework: How to use it, test using artificial faults, and develop new faults. http://wiki.apache.org/hadoop/HowToUseInjectionFramework.

[19] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd USENIX Symposium on Networked System Design and Implementation*, NSDI'05, 2005.

[20] C. Fu and G. B. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *29th International Conference on Software Engineering*, ICSE'07, pages 230–239, 2007.

[21] Google outage reportedly caused big drop in global traffic. http://www.cnet.com/news/google-outage-reportedly-caused-big-drop-in-global-traffic/.

[22] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[23] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: a framework for cloud recovery testing. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, NSDI'11, 2011.

[24] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, 2008.

[25] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 265–278, October 2011.

[26] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.

[27] Hadoop Distributed File System (HDFS) architecture guide. http://hadoop.apache.org/docs/stable/hdfs_design.html.

[28] Hadoop MapReduce. http://hadoop.apache.org/docs/stable/mapred_tutorial.html.

[29] Hadoop market is expected to reach usd 20.9 billion globally in 2018. http://www.prnewswire.com/news-releases/hadoop-market-is-expected-to-reach-usd-209-billion-globally-in-2018-transparency-market-research-217735621.html.

[30] DB-Engines ranking of wide column stores. http://db-engines.com/en/ranking/wide+column+store.

[31] HBase bug report 10452 – Fix bugs in exception handler. https://issues.apache.org/jira/browse/HBASE-10452.

[32] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notice*, 39(12):92–106, Dec. 2004.

[33] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM 2009 conference*, SIGCOMM '09, pages 243–254, 2009.

[34] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: finding liveness bugs in systems code. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation*, pages 243–256, April 2007.

[35] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity

multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pages 155–166, 2010.

[36] J.-C. Laprie. Dependable computing: concepts, limits, challenges. In *Proceedings of the 25th International Conference on Fault-tolerant Computing*, FTCS'95, pages 42–54, 1995.

[37] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation*, OSDI'14, 2014.

[38] S. Li, T. Xiao, H. Zhou, H. Lin, H. Lin, W. Lin, and T. Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proc. International Conference on Software Engineering (ICSE 2013), Software Engineering in Practice (SEIP) track*, May 2013.

[39] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation*, OSDI'04, 2004.

[40] G. C. Lorenzo Keller, Prasang Upadhyaya. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings International Conference on Dependable Systems and Networks*, DSN'08, 2008.

[41] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 31–44, 2013.

[42] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS'08, pages 329–339, 2008.

[43] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 conference*, SIGCOMM '02, pages 3–16, 2002.

[44] P. D. Marinescu, R. Banabic, and G. Candea. An extensible technique for high-precision testing of recovery code. In *Proceedings of the 2010 USENIX annual technical conference*, USENIX ATC'10, 2010.

[45] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 29(4):11:1–11:38, Dec. 2011.

[46] Missing dot drops Sweden off the internet. `http://www.networkworld.com/community/node/46115`.

[47] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution effiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.

[48] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[49] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, OSDI'04, 2004.

[50] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, USITS'03, pages 1–15, 2003.

[51] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 305–318, 2011.

[52] A. Rabkin and R. Katz. How Hadoop clusters break. *Software, IEEE*, 30(4):88–94, 2013.

[53] DB-Engines ranking of key-value stores. `http://db-engines.com/en/ranking/key-value+store`.

[54] Redis: an open source, advanced key-value store. `http://redis.io/`.

[55] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 270–280, 2009.

[56] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.

[57] C. Spatz. Basic statistics, 1981.

[58] M. Sullivan and R. Chillarege. Software defects and their impact on system availability — A study of field failures in operating systems. In *Twenty-First International Symposium on Fault-Tolerant Computing*, FTCS'91, pages 2–9, 1991.

[59] Summary of the Amazon EC2 and RDS service disruption. `http://aws.amazon.com/message/65648/`.

[60] The curse of the swallowed exception. `http://michaelscharf.blogspot.ca/2006/09/dont-swallow-interruptedexception-call.html`.

[61] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, 2011.

[62] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 193–204, New York, NY, USA, 2010. ACM.

[63] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the 24th ACM Sympo-*

sium on Operating Systems Principles, SOSP '13, 2013.

[64] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, 2009.

[65] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228, April 2009.

[66] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, 2011.

[67] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation*, OSDI'12, pages 293–306, 2012.

Shielding applications from an untrusted cloud with Haven

Andrew Baumann Marcus Peinado Galen Hunt
Microsoft Research

Abstract

Today's cloud computing infrastructure requires substantial trust. Cloud users rely on both the provider's staff and its globally-distributed software/hardware platform not to expose any of their private data.

We introduce the notion of shielded execution, which protects the confidentiality and integrity of a program and its data from the platform on which it runs (i.e., the cloud operator's OS, VM and firmware). Our prototype, Haven, is the first system to achieve shielded execution of unmodified legacy applications, including SQL Server and Apache, on a commodity OS (Windows) and commodity hardware. Haven leverages the hardware protection of Intel SGX to defend against privileged code and physical attacks such as memory probes, but also addresses the dual challenges of executing unmodified legacy binaries and protecting them from a malicious host. This work motivated recent changes in the SGX specification.

1 Introduction

Although users of cloud computing infrastructure may expect their data to remain confidential, today's clouds are built using a classical hierarchical security model that aims only to protect the privileged code (of the cloud provider) from untrusted code (the user's virtual machine), and does nothing to protect user data from access by privileged code. As a result, besides the hardware used to execute their applications, the cloud user must trust: (i) the provider's software, including privileged software such as a hypervisor and firmware but also the provider's full stack of management software; and (ii) the provider's staff, including system administrators but also those with physical access to hardware such as cleaners and security guards. Furthermore, as the Snowden leaks demonstrate [18, 19], the cloud user also implicitly trusts (iii) law enforcement bodies in any jurisdiction where their data may be replicated. By any measure, this is a large and inscrutable trusted computing base, and the related concerns are a significant factor limiting cloud adoption [13, 43].

The current best practice for protecting secrets in the cloud uses hardware security modules (HSMs) [e.g., 1]. These dedicated appliances rely on tamper-proof hardware to protect critical secrets, such as keys, and support a range of cryptographic functions, but come at a significant cost, and do not usually run general-purpose applications. Typical deployments use HSMs to protect key material, but transiently decrypt data on untrusted nodes for computation, rendering the data vulnerable to the threats outlined above. Previous research relied on trusted hypervisors to protect an application from a malicious OS [11, 25, 54, 60, 63], but cannot protect against a hypervisor controlled by a malicious or compromised cloud provider. Finally, although some applications can operate on encrypted data [4, 46, 55], cryptographic schemes for general-purpose computing [20, 21] have severe performance limitations.

Our objective is to run existing server applications in the cloud with a level of trust and security roughly equivalent to a user operating their own hardware in a locked cage at a colocation facility. Like the colocation provider (who is responsible only for power, cooling and network connectivity), the cloud provider is limited to offering raw resources: processor cycles, storage, and networking; it can deny service, but cannot observe or modify any user data except what is transmitted over the network. We refer to this property as *shielded execution*, and define it in §2. Essentially the inverse of sandboxing, it protects the confidentiality and integrity of code and data from an untrusted host. The high-level guarantee to the user is that secrecy is always preserved, and if their program executes, it behaves as if it ran on reference hardware under the user's control. The provider retains control of resource allocation, and may protect itself from a malicious guest.

Our prototype, *Haven*, implements shielded execution of unmodified Windows applications. It leverages Intel software guard extensions (SGX) [28, 29, 41], a set of new instructions and memory access changes summarised in §3.1. SGX allows a process to instantiate a secure region of address space known as an *enclave*; it then protects execution of code within the enclave, even from malicious privileged code or hardware attacks such as mem-

ory probes. While SGX was designed to enable new trustworthy applications to protect specific secrets by placing portions of their code and data inside enclaves [24], Haven aims to shield *entire unmodified legacy applications* written without any knowledge of SGX. This leads to two key challenges. First, executing legacy binary code inside an enclave pushes the limits of the SGX execution model: our target applications are large, raise and handle exceptions, dynamically allocate memory, and may execute arbitrary x86 instructions. Second, the code we seek to protect was written assuming that the OS it ran on would operate correctly, but the host OS may be malicious. To defeat such "Iago attacks" [10], where a malicious OS subverts a protected application by exploiting the application's reliance on correct results of system calls, we use an in-enclave library OS (LibOS). The LibOS used by Haven is derived from Drawbridge [47]; it implements the Windows 8 API using a small set of primitives such as threads, virtual memory, and file I/O. As we describe in §4, Haven implements these primitives using a mutually-distrusting interface with the host OS. This ensures shielded execution of unmodified applications; a malicious host cannot trick an application into divulging its secrets nor executing incorrectly. Combined with a remote attestation mechanism [2], Haven gives the user an end-to-end guarantee of application security without trusting the cloud provider, its software, or any hardware beyond the processor itself.

We developed Haven on an instruction-accurate SGX emulator provided by Intel, but evaluate it using our own model of SGX performance. Haven goes beyond the original design intent of SGX, so while the hardware was mostly sufficient, it did have three fundamental limitations for which we proposed fixes (§5.4). These are incorporated in a revised version of the SGX specification [29], published concurrently by Intel.

The contributions of this paper are:

- We define the concept of shielded execution (§2), describe how SGX supports it (§3.1), and later outline generalised hardware requirements (§7.4).
- We present Haven, the first system to implement shielded execution of unmodified binaries for a commodity OS, achieving mutual distrust with the entire host software stack (§4–5). Haven shields applications using mechanisms such as private scheduling, distrustful virtual memory management, and an encrypted and integrity-protected file system.
- We evaluate Haven's performance using unmodified server applications: SQL Server and Apache (§6).
- We identify minimal changes to SGX to enable efficient shielded execution of unmodified applications (§5.4), and note optimisation opportunities (§7.3).

2 Security Overview

2.1 Shielded execution

Like others [41, 44, 59], we use the term *isolated execution* to refer generally to mechanisms that protect the confidentiality and integrity of specific code and data from other actors. In contrast to previous protection mechanisms such as process isolation, sandboxing, managed code, etc. which serve to confine an untrusted program and protect the rest of a system from its actions, isolated execution refers to the inverse: *protecting specific code from the rest of the system, however large or privileged.*

Various forms of isolated execution are possible. Software implementations rely on a trusted component such as a hypervisor that implements isolation (e.g., using page protection and/or encryption) [11, 14, 25, 39, 54, 60, 63]. Conversely, in pure-hardware implementations, no software other than the isolated code is in the trusted computing base. While several hardware isolation mechanisms exist [9, 31, 34, 44, 59], SGX is the first commodity hardware that permits efficient multiplexing among multiple isolated programs without relying on trusted software.

However, isolation alone is not sufficient to protect applications. In order to be useful, an isolation mechanism must permit interaction with untrusted software or hardware, to communicate results or access system services, and it is at these points that a naïve isolated program is vulnerable [10]. For example, SGX isolates self-contained sequences of x86 instructions (typically, individual modules or functions) that are aware of the SGX protection model and are explicitly written to defend against threats outside the enclave, that do not handle faults or exceptions, and do not interact with the OS.

Shielded execution builds on an isolation mechanism to provide higher-level security properties; specifically, for an abstract program, it guarantees:

- *Confidentiality:* The execution of the shielded program appears as a "black box" to the rest of the system. Only its inputs and outputs, but no intermediate states, are observable.
- *Integrity:* The system cannot affect the behaviour of the program, except by choosing not to execute it at all or withholding resources (denial of service attacks). If the program completes, its output is the same as a correct execution on a reference platform.

While the term may be new, the underlying concept is not. In using a new term, we attempt to generalise beyond specific implementations such as cloaking [11, 12], "pieces of application logic" [33, 38, 39], protected modules [45] and high-assurance processes [25] that provide

shielded execution under a specific set of constraints. Moreover, in Haven, our goal is to relax those constraints to achieve shielded execution for unmodified application binaries with complex OS dependencies.

Note that shielded execution is necessary but not sufficient to meet our goal of confidential execution in the cloud. We also require an attestation mechanism to establish confidence in the integrity of a remote shielded program, and a mechanism to provision secrets directly to it, allowing it to operate on encrypted inputs and extend confidentiality beyond the confines of the shield mechanism. We describe how SGX supports this in §3.1.

2.2 Threat model and assumptions

We seek to protect the confidentiality and integrity of a user's unmodified server application from an untrusted cloud provider. We specifically exclude software-based isolation mechanisms, because besides their vulnerability to simple hardware attacks, we wish to give the cloud provider the unfettered ability to patch and update its privileged software (we expand on this later in §8). We therefore assume a powerful adversary that controls most of the provider's hardware and all its software.

At the hardware level, we assume that the processor itself is implemented correctly, and not compromised (so the adversary cannot extract secrets residing within it). The adversary has full control beyond the physical package of the processor, including memory and all I/O devices. They may probe memory, and arbitrarily alter or inject I/O including network traffic.

The adversary also controls the cloud provider's entire software stack, including the host OS, hypervisor, management software, platform firmware, BIOS, code executing in system management mode, and device firmware. As a result, they may interrupt execution of the user's program indefinitely, and may pass arbitrary values across the isolation boundary (e.g., the SGX enclave), including the results of calls to OS services and arbitrarily-injected upcalls. We assume a secure source of random numbers (which recent processors provide). However, the adversary may interfere with other sources of non-determinism such as thread interleaving, subject to the constraints of the hardware specification (e.g., the memory model).

We do not consider any side-channel attacks. Common side-channels, such as timing and cache-collision, have known (but expensive) attack mitigations [e.g., 8] that can be implemented by application software; others, such as power analysis, require hardware modifications, and are ultimately a limitation of our approach.

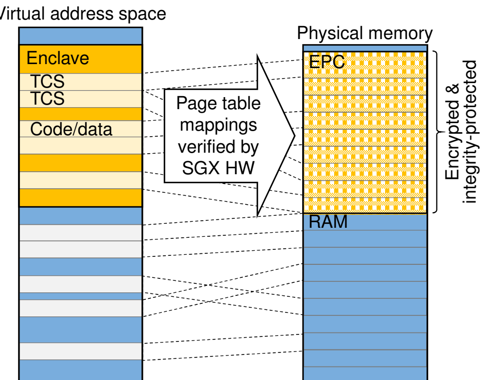


Figure 1: SGX virtual and physical memory layout

3 Background

3.1 Intel SGX

In this section, we summarise the SGX functionality relevant to Haven; readers are directed to the specification [28, 29, 41] for full details. Although SGX protects against any malicious privileged code (OS, hypervisor, firmware, system management mode, etc.), we refer to it collectively as simply the "OS".

Memory protection SGX protects the confidentiality and integrity of pages in an *enclave*, a region of a user-mode address space (Figure 1). While cache-resident, enclave data is protected by CPU access controls (the TLB). However, it is encrypted and integrity protected when written to memory, and if the data in memory is modified, a subsequent load will signal a fault.

SGX mediates page mappings at enclave setup and maintains shadow state for each page. Enclaves are created by an `ECREATE` instruction, which initialises a control structure in protected memory. Once an enclave has been created, pages of memory are added to it using `EADD`. These pages are allocated by the OS, but must occupy a specific region of physical memory: the enclave page cache (EPC). For each EPC page, hardware tracks its type, the enclave to which it is mapped, the virtual address within the enclave, and permissions (read, write, execute). On each enclave page access, after walking the page table, SGX ensures that the processor is in enclave mode, the page belongs to the EPC and is correctly typed, the current enclave maps the page at the accessed virtual address, and the access agrees with the page permissions.

Like the RAM backing it, EPC is a limited resource. Therefore, SGX enables the OS to virtualise EPC by paging its contents to other storage [28, §3.5]. Privileged instructions cause the hardware to free an EPC page cho-

sen by the OS, writing its contents to an encrypted buffer in main memory, which the OS may then relocate. To prevent rollback attacks on page-in, the hardware keeps a version number for the page in EPC. It also requires the OS to follow a hardware-verified protocol to ensure that TLB shootdown has completed when evicting a page.

Attestation SGX supports CPU-based attestation [2], enabling a remote system to verify cryptographically that specific software has been loaded within an enclave, and establish shared secrets allowing it to bootstrap an end-to-end encrypted channel with the enclave.

During enclave creation, a secure hash known as a *measurement* is established of the enclave's initial state. The enclave may later retrieve a *report* signed by the processor that proves its identity to, and communicates a unique value (such as a public key) with, another local enclave. Using a trusted *quoting enclave*, this mechanism can be leveraged to obtain an attestation known as a *quote* which proves to a remote system that the report comes from an enclave running on a genuine SGX implementation [2]. Ultimately, the processor manufacturer (e.g., Intel) is the root of trust for attestation.

Enclave entry and exit Besides protecting the content and integrity of memory mappings, SGX also mediates transitions into and out of the enclave, and protects the enclave's register file from OS exception handlers. This is managed using a thread control structure (TCS).

User code begins executing an enclave by invoking `EENTER` on an idle TCS; this acts as a call gate, transferring control to a defined entry point within the enclave. Enclave code may access enclave pages according to the protection model outlined above; it may also read and write memory outside the enclave region (as permitted by OS page tables), but any attempt to execute code there faults. The processor continues in enclave mode until software explicitly leaves it by invoking `EEXIT`, or until an interrupt or exception returns control to the OS, which is known as an *asynchronous exit*. After an explicit exit, control resumes outside the enclave at an address chosen by the enclave; in this way `EENTER` and `EEXIT` can be used with stubs that wrap invocations of enclave functions, taking care to validate inputs on entry and scrub secrets on exit from any registers not used as return values.

After an asynchronous exit, control transfers to the OS exception handler; typically this would save the registers for later use (e.g., when next scheduled), but the OS cannot be trusted with the enclave's register state. Instead, SGX saves the full context and information about the cause of the exit in the TCS, replacing it with a synthetic context before reporting the exception to the OS. The enclave may later be resumed by `ERESUME` on the TCS, which restores its last saved context. Alternatively, the OS can re-enter the enclave, giving it the opportunity to inspect and modify its own state before resuming; this is used to report an exception which must be handled by the enclave.

SGX is an imperfect implementation of shielded execution according to our definition in §2.1, because the OS exception handler observes some of the enclave's internal state: the exception vector, and in the case of a page fault, the type of access and base address of the page [28, §4.4]. This allows the OS to retain control over resource management (i.e., CPU time and memory); in general, it can deny service to the enclave, but cannot cause it to execute incorrectly. We discuss hardware designs to decouple resource management from observations of guest behaviour later in §7.4.

Dynamic memory allocation As described, SGX does not allow enclave pages to be added after creation, nor EPC permissions changed, which is clearly insufficient for Haven to run unmodified applications. However, revision 2 of the SGX specification [29] includes new instructions allowing the enclave and host OS to *cooperatively* add/remove enclave pages and modify their permissions.

Allocation requires cooperation, because the host manages EPC but cannot be trusted to arbitrarily add enclave pages (e.g., in an unallocated region). To allocate a new page, the host invokes `EAUG` to place an unused EPC page at a specific offset in an enclave; this must then be acknowledged by the enclave executing `EACCEPT`, before it becomes accessible. Similarly, reducing permissions or removing pages also requires cooperation, because like page eviction, hardware must help ensure TLB shootdown has occurred. SGX includes instructions (`EMODT`, `EMODPR`, `EBLOCK`, `ETRACK` and `EACCEPT`) to enable this.

These operations do not change the enclave measurement established by `EINIT`; since the modified pages come under the full control of the enclave, its identity is equivalent for trust purposes.

3.2 Drawbridge

Haven builds on Drawbridge [6, 47], a system supporting low-overhead sandboxing of Windows applications. Drawbridge consists of two core mechanisms, both of which Haven leverages: the picoprocess, and library OS.

The *picoprocess* is a secure isolation container constructed from a hardware address space, but with no access to traditional OS services or system calls [15]; instead, a narrow ABI of OS primitives is provided, implemented using a security monitor. The ABI consists of 40 downcalls and three upcalls [6]. Downcalls are requests for OS services including virtual memory, thread-

ing, and I/O streams (e.g., files and network sockets). Upcalls are initiated by the host, have only input parameters, and do not return; they are used for initialisation, thread startup, and exception delivery. In Haven, as in Drawbridge, the picoprocess serves to protect the host (i.e., the cloud provider) from a potentially-malicious guest.

The Drawbridge *LibOS* is a version of Windows 8 refactored to run as a set of libraries within the picoprocess, depending only on the ABI. It consists of lightly-modified binaries for most user-mode and some kernel components of Windows, and a "user-mode kernel" that implements the interfaces on which they depend.

Together, the picoprocess and LibOS enable sandboxing of unmodified Windows applications with comparable security to virtual machines, but substantially lower overheads. While Drawbridge aims only to protect the host from an untrusted guest, Haven shields the execution of the application and LibOS from an untrusted host, thereby enabling mutual distrust between host and guest.

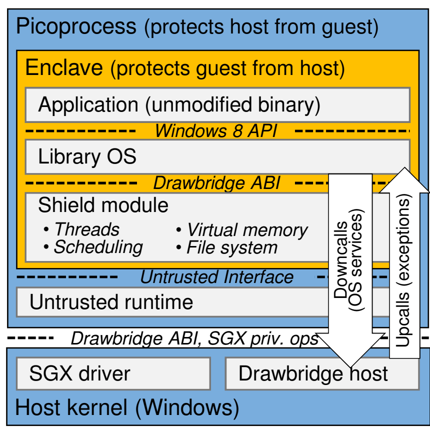


Figure 2: Haven components and interfaces

4 Design

We now present the design of Haven, which leverages the instruction-level isolation mechanism of SGX to achieve shielded execution of entire legacy application binaries. In doing this, we address two key challenges: protecting from a malicious host OS, and executing existing binaries in an enclave. We first discuss these in more detail.

4.1 Design challenges

Malicious host OS A general class of threats known as Iago attacks arises when a malicious OS attempts to subvert an isolated application by exploiting its assumption of correct OS behaviour, for example when using the results of system calls [10]. Besides simply returning semantically-incorrect results from system calls (e.g., returning the address of an already-allocated region for a new memory allocation), the malicious OS may seek to exploit latent bugs in the application. For example, it may allocate valid but abnormally-high virtual addresses, return unusual values for parameters such as memory size and number of processors, alter timing to seek to exploit latent race conditions, inject spurious exceptions, return unexpected error codes from system calls, or simply fail calls that an application naively assumes will succeed.

Our approach to this challenge is twofold. First, we limit its scope using a LibOS within the enclave. The LibOS implements the full OS API using a much narrower set of core OS primitives. Since the LibOS is under user control, and can be arbitrarily tested or inspected offline, we assume that it is not malicious (to the user), even though it may be large, complex, and contain bugs. Second, having reduced the scope of attacks by narrowing the interface they must traverse, we use established techniques to correctly implement the OS primitives in the presence of a malicious host: careful defensive coding, exhaustive validation of untrusted inputs, and encryption and integrity protection of any private data exposed to untrusted code.

Unmodified binaries SGX was designed to protect limited subsets of application logic [24], however full application binaries have properties that make them challenging to execute in an enclave. They load code and data at runtime, dynamically allocate and change protection on virtual memory, execute arbitrary user-mode instructions (including some not supported by SGX), raise and handle exceptions (e.g., page faults, divide-by-zero or floating-point exceptions), and use thread-local storage.

Haven addresses each of these challenges. For some, such as thread-local storage, we rely on enhancements to SGX described later in §5.4. For most, we work around the limitations, by emulating unsupported instructions, carefully validating and handling exceptions that occur within an enclave, and modifying LibOS behaviour.

4.2 Architecture

Figure 2 shows the architecture of Haven. We create an enclave within the Drawbridge picoprocess containing the entire application and LibOS. To protect the LibOS and application from a malicious host, Haven augments Drawbridge with two layers: a *shield* module below the LibOS in the enclave, and an *untrusted runtime* outside the en-

Upcalls:

```
ExceptionDispatch(ExceptionInfo)
ThreadEntry()
```

Downcalls:

```
AsyncCancel(AsyncHandle)
AsyncPoll(AsyncHandle) -> Results
DebugStringPrint(Message)
EventClear(EventHandle)
EventSet(EventHandle)
ObjectClose(Handle)
ObjectsWaitAny(Count, Handles, Timeout) -> Index
ProcessExit(ExitCode)
StreamAttributesQueryByHandle(StreamHandle) -> Attribs
StreamFlush(StreamHandle)
StreamGetEvent(StreamHandle, EventId) -> EventHandle
StreamOpen(URI, Options) -> StreamHandle
StreamRead(StreamHandle, Off, Len, Buf) -> AsyncHandle
StreamWrite(StreamHandle, Off, Len, Buf) -> AsyncHandle
SystemTimeQuery() -> Time
ThreadCreate(Tcs) -> ThreadHandle
ThreadExit()
ThreadInterrupt(ThreadHandle)
ThreadYieldExecution()
VirtualMemoryCommit(Addr, Size, Prot)
VirtualMemoryFree(Addr, Size)
VirtualMemoryProtect(Addr, Size, Prot)
```

Figure 3: Untrusted interface to enclave.

clave. These effectively interpose on the LibOS/host interface (the Drawbridge ABI) [6], implementing a shielded version in the enclave by calling out to the untrusted host.

Our design is independent of the specific LibOS; recent Linux LibOSes [6, 27, 58] might also be used.[1]

Shield module As with all code inside the enclave, this is in the application's trusted computing base. Its high-level role is to implement the Drawbridge ABI required by the LibOS in terms of a more limited subset of core OS operations. It therefore includes private implementations of typical kernel functionality such as memory management, a file system, and thread synchronisation. It also acts as a trusted bootloader for the LibOS and application.

The shield is responsible for protecting the LibOS and application from Iago attacks outside the enclave. It does this by careful validation of all parameters and results passed across a narrow interface with the untrusted runtime. At its most basic, this validation consists of ensuring that the parameters of upcalls and results of downcalls are consistent with their specification. For example, the number of bytes read from a stream cannot be more than the requested size, and it is not acceptable to return an error code indicating a timeout for an operation that cannot do so (more generally, each downcall has a specific list of acceptable failure codes). Specific calls require further validation, using either hardware support (e.g., when changing virtual memory permissions), or additional software (e.g., for thread synchronisation), and are discussed in the relevant sections below.

Since our threat model permits denial of service, the shield can and does handle any incorrect host behaviour by panicking: it emits a short debug message, requests the host to terminate its process, and rejects subsequent attempts to enter the enclave.

Untrusted interface The interface at the enclave boundary must allow the shield to verify the correctness of all operations while also enabling an efficient implementation. Besides minimality, our guiding principle in designing it was a form of policy/mechanism separation [32]: *the guest controls policy for virtual resources (virtual address allocation, threads, etc.), while the host manages policy only for physical resources (e.g., memory and CPU time)*. In general, this prevents operations that give any implementation freedom to the host beyond physical resource allocation, makes verification efficient, and limits the scope of attacks.

The interface is summarised in Figure 3; it is expressed as a Drawbridge ABI subset, with fewer (22 rather than 40) calls and fewer permissible arguments; specifically:

- calls to commit, free and protect specific pages;
- thread management and signalling;
- I/O streams to access untrusted storage and network;
- a source of system time.

Untrusted runtime Primarily bootstrap and glue code, this is trusted by neither enclave nor host kernel. Its main tasks are creating the enclave, loading the shield, and forwarding calls between the enclave and host OS.

4.3 Shield services

Virtual memory The virtual address region occupied by a Haven enclave always starts at zero (enforced by a check at startup), allowing the enclave to reliably detect and handle NULL pointer dereferences. Otherwise, a malicious host OS could map pages there, and redirect NULL accesses to data of their choosing. The enclave's virtual size must be large enough for all possible allocations by

[1] We note however that `fork()` would be both complex and expensive, as it requires a new enclave communicating via untrusted channels.

the application/LibOS, and small enough to leave some address space for the untrusted runtime and host OS. In our prototype, enclaves occupy 64GB of address space.

The shield manages virtual memory within the enclave. It includes a region allocator, tracking sub-regions of the enclave that are reserved for use. For each reserved region, it tracks which pages are committed (i.e., accessible to the application), and for those pages, their permissions (read, write, execute). For each allocation, the shield chooses an address based on its knowledge of allocated regions. When memory is committed or its protection is changed, the shield calls out to the host to make the appropriate changes (e.g., allocating and mapping EPC pages and performing TLB shootdown if necessary), then uses the dynamic memory allocation instructions described in §3.1 to ensure that the expected changes were made. To prevent exploits of latent bugs in the application or LibOS, the shield never allows the host to choose virtual addresses. It also blocks the application from using non-enclave memory, by failing requests to allocate it.

Storage While SGX provides confidentiality and integrity protection for data in memory, Haven must also support secure persistent storage. Rather than simply encrypting file contents, which risks leaking guest state through file metadata, the shield implements a private filesystem. Our prototype uses a FAT32 filesystem inside an encrypted virtual hard disk (VHD) image.

The shield encrypts each disk block independently with an authenticated encryption algorithm (AES-GCM [40]), keying the encryption to the block number. Like other systems [16, 17, 26, 36, 62], a Merkle tree [42] protects the integrity of the overall disk. This can be implemented with little overhead, as only the root and the leaf nodes of the tree are persisted to disk [26]. Like InkTag [25], we store the crypto metadata (message authentication codes of data blocks, nonces, and the Merkle tree root) in separate blocks from filesystem data. We also adapted InkTag's two-hash-versions scheme to maintain consistency after crashes. We discuss rollback attacks in §7.2.

Threads and synchronisation To prevent the host from exploiting the application, for example by allowing two threads to concurrently acquire a mutex, the shield implements a form of user-level scheduling [3, 37]. At startup, it creates a fixed number of threads according to the desired level of parallelism (typically, the number of hardware threads). These operate as virtual CPUs supporting an arbitrary number of application threads inside the enclave. Besides multiplexing application threads across the virtual CPUs, the shield's scheduler implements primitives for events, mutexes and semaphores. It maintains its internal state (run queues and synchronisation objects) using atomic instructions for safety, and uses the untrusted interface's event and interrupt mechanisms to support suspending/resuming and signalling the virtual CPUs.

Table 1: Summary of component sizes

	LoC[a]	Size
Drawbridge LibOS	millions[b]	209 MB[c]
Shield module	23,095	180 kB
Untrusted runtime	7,446	52 kB
SGX driver	4,520	41 kB

[a] Lines of code counted by David A. Wheeler's "SLOCCount".
[b] See Porter et al. [47] for a breakdown (of a previous version).
[c] We report file size for all binaries in the LibOS; the subset that is loaded depends on the application, but is usually much smaller.

The untrusted host can deny service by delaying wake-ups or interrupts, but cannot cause the application to execute incorrectly. Moreover, its ability to exploit latent race conditions in the application by delaying guest execution is severely curtailed by the multiplexing of application threads (which it cannot observe) onto virtual CPUs.

Miscellaneous The shield handles calls for entropy generation (using `RDRAND`, a secure source of randomness) and dynamic loading/relocation of application binaries. Our loader does not yet implement address-space layout randomisation [7]; this is planned for future work.

Process creation is not supported. While not inconceivable, it would be extremely complex and expensive to implement on SGX, requiring the creation of a new enclave (in a separate address space), and communication with it over untrusted channels. One advantage of the Windows OS is that surprisingly few applications use child processes [6, 47]. For those that do, it is often sufficient to run the "subprocess" in a different portion of the parent's address space (i.e., in the same enclave), since the API has no `fork()` operation; this is supported by the LibOS.

5 Implementation

Table 1 reports the size of various components in our current prototype. Besides implementing the shield and untrusted runtime, we added SGX support to our host OS (Windows 8.1) by writing a driver and making some kernel changes. The driver implements SGX kernel-mode operations: allocating and mapping EPC pages, and creating and destroying enclaves. It is trusted by the host, but untrusted by enclave code. We also modified the host kernel to enable efficient mapping of EPC pages to user-mode. This was necessary, because EPC regions appear as reserved device space to the kernel, and the existing

driver APIs for mapping device memory to user-mode did not anticipate the need for efficient page-granular mapping and protection changes. We also implemented support for debugging an enclave using the SGX debug mechanisms [28, Chapter 7]. Finally, we made minor modifications (249 lines of code) to the LibOS to avoid using shared memory within the picoprocess,[2] which SGX does not support.

5.1 Application deployment and attestation

Haven applications are deployed similarly to cloud VMs, with an extra attestation step we now describe. A user constructs a disk image containing application and LibOS binaries and data, and then encrypts it symmetrically, retaining the key. The encrypted VHD and shield binary are sent to the cloud provider. The shield is not encrypted, but its integrity will be verified.[3] The cloud provider establishes a picoprocess, and loads the untrusted runtime, which then creates an enclave and loads the shield module. While the shield is loaded, the SGX hardware attestation mechanism (§3.1) is used to measure (i.e., compute a secure hash of) its code and initial state. The shield receives two startup parameters: a structure of untrusted parameters chosen by the host, such as addresses of downcall functions in the untrusted runtime, and trusted parameters chosen by the user, such as configuration options and environment variables, which form part of the enclave's measurement.

After initialising itself, the shield generates a public/private key pair, and then uses its parameters to establish a network connection with the user – this may be a machine physically under the control of the user, or another enclave in the cloud. In either case, the shield uses the SGX attestation mechanism to produce a quote containing its public key which it sends to the user, proving that it has been correctly loaded and executes in an SGX enclave. If the enclave's measurement is as expected (i.e., if the shield was loaded correctly with the desired parameters), the user encrypts the VHD key using the public key contained in the quote, and sends it back to the shield; any tampering with the shield binary is detected and subverted at this point. Assuming it was loaded correctly, the shield may now decrypt the VHD key using its private key, and use it to access the contents of the VHD, allowing it to continue to load the LibOS and application.

[2]Code that relied on making multiple mappings of a shared memory section within the process was changed to use a single virtual address.

[3]If confidentiality of the shield was desirable, a smaller trusted bootloader could be used whose only task would be to perform attestation at startup and then decrypt and load the shield binary.

From this point onward, communication with the outside world, and therefore access to any secrets contained in the VHD, is under application control. Typical server applications supporting SSL-encrypted connections may be configured using certificates and keys stored directly in the VHD, and accessed over the cloud provider's untrusted network. For future work, we are planning to add support for encrypted virtual private networks between a user's enclaves (or trusted hosts), providing a secure network to applications that require one.

5.2 Enclave entry/exit

In Haven, an application performs most of its execution in the enclave, calling out to the untrusted host only for system services. This is the opposite of the typical SGX usage model of untrusted code calling into an enclave [24]. To perform an upcall, the untrusted runtime loads the upcall parameters into specific registers, and invokes `EENTER`, which does not return to its caller but instead delivers control to the shield entry point inside the enclave.

To perform a downcall, the shield passes arguments in registers while clearing any unused registers (to prevent leaking secrets), stores the return address and stack pointer inside the shield's thread record, and invokes `EEXIT` with a target address of the relevant downcall handler. SGX leaves the enclave and executes the untrusted handler, which first loads a stack pointer before calling C code. When the downcall returns (generally, after a system call) its results are delivered to the enclave by `EENTER`, which re-enters the enclave at the shield entry point.

The shield must disambiguate the different entry causes. To do so it inspects the SGX thread structure, which identifies whether an exception occurred, and its own thread record, which records whether a downcall was in progress. It then reloads the stack pointer, and either calls (on an upcall) or returns to (on a downcall) C code.

Parameters that are passed by reference (e.g., I/O buffers) cannot be located inside the enclave, since they are inaccessible to the host. Instead, the shield allocates a "bounce buffer" from a memory region outside the enclave, and copies the parameters appropriately. We are considering (but have not yet implemented) an optimisation to the file system to encrypt directly into the bounce buffer on writes, and decrypt from it on reads; this would reduce the copy overhead for most file I/O.

5.3 Exception handling

When a page fault occurs within an enclave, SGX saves the register context and fault information to an in-enclave data structure (see §3.1). It then delivers an exception

to the host OS, which may choose to handle the fault (e.g., by lazily updating a page table) and resume execution, or to report it to the user process. In the latter case, the untrusted runtime upcalls the shield in the enclave, which must then determine the true cause using the information and register context provided by SGX. The shield exception handler performs sanity-checks to ensure that the exception is valid and should be reported to the LibOS. These include checking that: an exception actually occurred (as reported by SGX); the instruction is in the enclave but not the shield module (which should never fault); and the fault type (read, write, or execute) is consistent with the page's expected permissions.

The shield prepares to deliver the exception to the LibOS, by copying the context and cause in a format defined by the Drawbridge ABI, and modifying the context to run the LibOS exception handler. Haven must now resume the modified context. Unfortunately, SGX only allows `ERESUME` outside an enclave, so the shield must `EEXIT` to a small (untrusted) stub that immediately resumes the enclave, restoring the context. We discuss the performance implications of this additional exit later, in §7.3.

Most other exceptions are handled similarly to page faults. The one special case is illegal instructions: as we describe later in §5.4, some user-mode instructions are illegal in an enclave. The trusted exception handler decodes and emulates these, by modifying the processor context and advancing the instruction pointer.

5.4 SGX limitations and workarounds

In addition to the need for dynamic memory allocation, we encountered three architectural limitations with SGX as initially specified [28] that made it impossible to run existing application and LibOS binaries. We summarise these issues, our workarounds, and proposed changes. Working with Intel, these changes are now incorporated in the revised SGX specification [29].

Exception handling SGX allows an enclave to handle its own exceptions by reporting the exception cause and register context securely in the TCS. However, while the registers are always saved, not all exception causes are reported to the enclave. For example, hardware interrupts are of no relevance to the enclave and may reveal private information about the host configuration, so they are not reported. As originally specified [28, §2.6.3], the list of reported exceptions included program faults such as division by zero, breakpoint instructions, undefined instructions, and floating point / SIMD exceptions, but not page faults or general protection faults. This prevented an enclave from handling these faults without trusting the host for information such as faulting address and access type. However, page faults are commonly handled by user-mode code, for example in demand loading or stack allocation. General protection faults are less common, but may also occur, e.g. in a LibOS emulating privileged instructions. SGX now reports these faults to the enclave [29].

Permitted instructions SGX disallows in-enclave execution of instructions that may cause a VM exit or change software privilege levels [28, §3.6]. Unfortunately, three of these instructions are commonly encountered in LibOS and application binaries: `CPUID`, `RDTSC`, and `IRET`.

`CPUID` This instruction queries processor features, generally to test for extended instructions. SGX prevents its use within an enclave, because a virtual machine may be configured to trap and emulate it, but emulation is impossible since the enclave's registers are not visible to the hypervisor. Instead, the processor signals an invalid instruction, and Haven's exception handler emulates `CPUID` using static knowledge of features available on SGX.

`RDTSC` and `RDTSCP` These instructions return the cycle counter, and are commonly used as a low-overhead time source, e.g. to measure hold-time in adaptive spinlocks. The initial version of SGX prevented their use because, like `CPUID`, they may cause a VM exit. However, unlike `CPUID`, they are not feasible to emulate: first, there is no reliable source of time, and second, most uses of `RDTSC` rely on its low overhead, which emulation cannot achieve. Instead, the SGX specification was revised to permit these instructions if VM exiting is disabled [29].

`IRET` Nominally an "interrupt return", this is used at the end of an exception handler when restoring processor state. It pops registers including instruction and stack pointers, and returns in the new context.[4] Since `IRET` can also change protection level, SGX disallows its execution in an enclave. Haven presently emulates `IRET`, but this adds overhead to exceptions, as we discuss later in §7.3.

Thread-local storage For legacy reasons, thread-local data on x86 is accessed via FS or GS segments. On SGX, `EENTER` and `ERESUME` load private FS and GS base addresses from the TCS. However, because a TCS is immutable once created, the addresses must be known at startup. Instructions exist to change FS/GS (`WRFSBASE`, `WRGSBASE`), but these could not reliably be used in an enclave, since the changes were not saved on asynchronous exits, and `ERESUME` restored FS and GS from the TCS.

As a result of this limitation, it was not possible to context-switch a TCS between application threads, and

[4] `IRET` is used since, to our knowledge, it is the only user-mode instruction that can restore a complete context, including volatile registers.

therefore impossible to perform user-mode scheduling within an enclave. As a workaround, the Haven prototype maps application threads 1:1 onto TCSs and host threads.[5] Although the shield's scheduler still ensures the correct behaviour of all synchronisation primitives, the host's ability to control scheduling of application threads makes it more likely that a malicious host could exploit application-level bugs by arbitrarily delaying threads.

This problem has been addressed in the revised SGX specification [29], but an implementation was not yet available to us, so our prototype relies on the workaround.

5.5 Unimplemented aspects

Our prototype implements the full design, with three exceptions. Rather than the attestation mechanism, we send the VHD key in the clear. We also built a simplified version of the disk integrity scheme that has equivalent performance, but cannot detect all block rollback attacks. Finally, we "emulate" `CPUID` by exiting the enclave to execute the instruction. We do not expect these shortcuts to materially impact performance.

6 Performance Evaluation

We developed and tested Haven using a functional emulator for SGX provided by Intel. However, in the absence of an SGX CPU or cycle-accurate emulator, we must do our own performance modelling. Our approach is to measure Haven's sensitivity to key SGX performance parameters.

In this section, we first describe our performance model, before reporting results for two typical cloud applications: Microsoft SQL Server, and Apache HTTP Server. Our performance experiments were run on a system comprised of a 4-core Intel Core i7-4700HQ CPU running at 2.4GHz with 8GB of 1600MHz DDR3 RAM, a 240GB SSD (Intel SC2CW240A3), and a gigabit Ethernet interface (Intel I217-LM) running Windows 8.1 Pro. We used this mobile-optimised platform, because it permits us to adjust the DRAM frequency and timings, allowing us to simulate a variable memory penalty for SGX.

6.1 Performance model

To model performance, we assume that an SGX implementation will perform the same as a current CPU, except for (i) additional costs (direct and indirect) of SGX instructions and asynchronous exits, and (ii) an additional memory penalty (latency and/or bandwidth of cache misses) for memory encryption when accessing EPC.

Many SGX instructions are executed only at enclave startup, and are therefore irrelevant to the performance of long-running server applications. We also assume that the EPC will be large enough to hold the working set of our applications, and therefore do not model the overhead of paging it to backing store. The only remaining direct overheads for Haven performance on SGX are the instructions for dynamic memory allocation (§3.1), and transitions into and out of enclave mode: `EENTER`, `EEXIT`, `ERESUME`, and asynchronous exits. The dynamic allocation instructions only check and update page protection metadata. The transitions are documented as requiring a TLB flush [28] and also perform a series of checks and updates.

For our evaluation, we implemented a second version of Haven that does not use SGX. Instead, it simulates SGX performance for the above critical instructions by busy-waiting for a configurable number of processor cycles, which we vary. In addition, for each enclave transition a system call is used to flush the TLB. Since the system call itself adds overhead not present on SGX, we view this as a conservative estimate for the performance of SGX. We cannot simulate the overhead of disallowed instructions such as `IRET` and `CPUID`, since there is no practical way to make them trap without SGX. However, we know from experience with the emulator that `CPUID` is only invoked at startup, and `IRET` is relatively rare (e.g., we observed around 100 `IRET`s per second for a web server workload).

Memory penalties for EPC access are more difficult to model. We simulate the impact of slower EPC by artificially reducing the system's DRAM frequency.

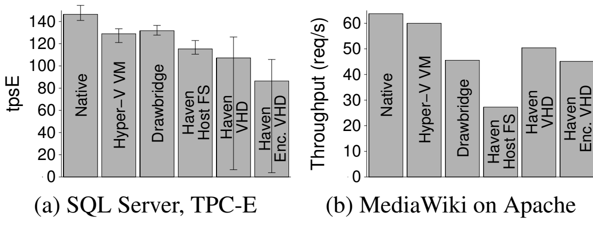
6.2 Application workloads

Database We run Microsoft SQL Server 2014, Enterprise Edition, and TPC-E [56], a standard online transaction processing benchmark. We use the default configuration for SQL Server when running natively or in a VM, but for Drawbridge and Haven we varied some parameters. Drawbridge does not support large pages or locked physical allocations, so we disabled them. We also limited the buffer cache to 6.5GB (the best-performing size), because the LibOS does not report physical memory usage, and the server's default behaviour led to excessive paging.

The TPC-E clients run on a single machine connected to our test system by a local gigabit network. We generated a database of 1000 customers[6] and left other pa-

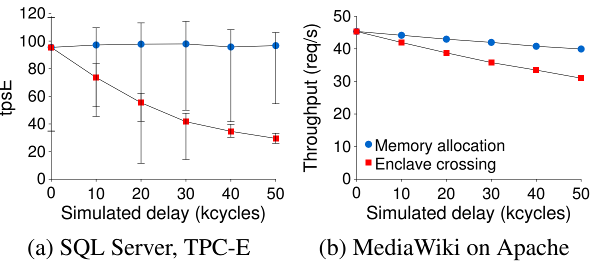
[5] This workaround is possible since the storage sizes for FS and GS are constant and Drawbridge does not choose their location [6, §3.1].

[6] Our database is smaller than the minimum for official TPC-E results (5000 customers), but sufficient to saturate our test system.



(a) SQL Server, TPC-E (b) MediaWiki on Apache

Figure 4: Performance breakdown



(a) SQL Server, TPC-E (b) MediaWiki on Apache

Figure 5: Sensitivity to SGX instruction overhead

rameters at the default settings. For each run we allowed at least 30 minutes of warm-up time, and then measured transaction performance for one hour, reporting the overall throughput. Error bars show minimum and maximum throughput over the run for a sliding 1-minute interval.

Web server We run Apache HTTP server[7] version 2.4.7 and PHP 5.5.11. We configured Drawbridge to run Apache's worker processes in the same address space (and enclave), and modified Apache's configuration to avoid using `AcceptEx`, which exposed a compatibility bug in the LibOS socket code. We installed MediaWiki 1.22.5 backed by a SQLite database, and enabled the Alternative PHP Cache for intermediate code and MediaWiki page data. We benchmarked the server using 50 worker threads on the client that repeatedly fetched the 14kB main page over persistent SSL connections for a period of 5 minutes.

6.3 Results

Overall performance We begin by comparing the performance of Haven to alternative host environments (none of which provide shielded execution). Figure 4 shows a performance breakdown for several configurations of each workload: native execution on Windows 8.1, in a Hyper-V VM, in Drawbridge, and three different configurations of Haven: one that trusts the host to implement the filesystem, the next using the private (VHD-backed) filesystem but not encrypting it, and finally the full system with VHD encryption and integrity protection enabled. In all Haven workloads, we flush the TLB on enclave crossings, but do not insert any additional delay for the SGX instructions. We verified that the server's CPU (and not network or storage I/O) is the bottleneck in all non-Haven runs, so these results give a reasonable indication of the overhead of the various software components.

For SQL Server, the extra runtime layers and TLB flush on enclave crossings give Haven a 13% slowdown vs. Drawbridge. Furthermore, Haven's unoptimised FAT filesystem is a bottleneck for the I/O-intensive SQL workload. Besides a further 25% slowdown (with encryption), it shows significant drops in throughput when limited I/O bandwidth causes the server to periodically delay transaction processing to allow checkpoint writes to complete.

Drawbridge and Haven exhibit relatively poor networking performance with Apache, because all socket operations traverse a security monitor in a separate process. Moreover, Haven performs substantially (40%) worse than Drawbridge with the host filesystem, because of many small file operations that flush the TLB. The private filesystem avoids this and even outperforms Drawbridge, since the workload is read-intensive and served almost entirely from the buffer cache inside the enclave.

Sensitivity to SGX instruction overhead Figure 5 shows the sensitivity of our workloads to SGX overheads. We vary the delay for either dynamic memory management instructions or enclave crossings while keeping the other at zero; the TLB is flushed on enclave crossings in all cases. SQL Server is sensitive to crossing overhead, with diminishing effects beyond 30k cycles, but unaffected by memory allocation overhead because few allocations occur in its steady state. The web server's throughput is sensitive to both parameters, because memory is allocated in request handlers, but drops less overall.

Sensitivity to EPC performance We artificially reduced the memory performance of our system, by lowering the DRAM clock rate from 1600MHz to 1067MHz.[8] This slowed the memory system by a third overall, but only reduced TPC-E throughput by 21%, and web throughput by 7%. We conclude that memory-intensive workloads are sensitive to EPC performance, but note that our experiment over-estimates its effect, since only some of the system's memory accesses would go to EPC.

Summary For now, we must speculate about performance of SGX implementations, but find our results encouraging: for large, complex, CPU and memory-intensive applications such as SQL Server, and for OS-intensive applications like a modern web stack, even given

[7] We used the 64-bit VC11 build from www.apachelounge.com.

[8] We reduced the DRAM clock multiplier, but kept the delay times (such as CAS latency, expressed in clock cycles) unchanged.

our inefficient prototype and assuming 10,000+ cycles for SGX instructions, Haven's performance penalty vs. a VM is 31–54%. We suspect that significant classes of users will readily accept such overheads, in return for not needing to trust the cloud.

7 Discussion

This section discusses various issues, starting with an analysis of the trusted computing base. We then cover future work, suggest SGX optimisations, and discuss general hardware support for shielded execution.

7.1 Trusted computing base

As Table 1 shows, the trusted computing base (TCB) of Haven is substantial, because the LibOS includes a large subset of Windows. However, in contrast to the current cloud model, all code in the enclave, and thus all code in the user's TCB, is under user control. They may use any means to achieve trust, including scanning for malware, code inspection, etc., and update it at will.

Ultimately, our goal is not to minimise the TCB, but rather to give the user equivalent trust in the confidentiality and integrity of their data when moving an application from a private data centre to a public cloud. In this regard, Haven addresses two real threats: a malicious employee of the cloud provider with either admin privileges or hardware access, or a government subpoena.

Besides the software TCB, Haven also relies on the processor's correctness. While a feature like SGX undoubtedly adds complexity, hardware (even microcode) is extremely hard for an attacker to modify, and hardware vendors perform significant validation to ensure correctness.

7.2 Future work

Storage rollback Haven does not currently prevent rollback of filesystem state beyond the enclave's lifetime. It cannot avoid the following attack: the enclave is terminated (e.g., the host fakes a crash), and its in-memory state is lost. A new instance of the enclave accessing the VHD is guaranteed to read consistent data, but not necessarily the latest version. Protecting against such attacks requires secure non-volatile storage [45]. Such storage may be located on other nodes, but the cost of network communication on every write is likely prohibitive. Instead, we plan to communicate only on "critical" writes (e.g., transaction commits) to balance this cost against the likely risks.

Untrusted time Our prototype relies on the host for system time and timeouts. However, a malicious host may lie about the time or signal timeouts early. We are planning two mitigations. One is to ensure the clock always runs forward. The other uses the cycle counter as an alternative time source; after calibrating it via network time synchronisation, we can check for early timeouts.

Cloud management Besides isolation, virtual machines can be saved, resumed and migrated. However, the implementation of these features depends on the host's ability to capture and recreate guest state, something that Haven explicitly prevents. We aim to support similar features cooperatively, using prior work that implemented checkpoint and resume at the Drawbridge ABI level [6]. In its simplest form, the host could request the Haven guest to suspend itself, which it would do by capturing its own state to an encrypted image. The host may then establish a new enclave to resume execution on another node. Before gaining access to the encrypted image, the new guest would perform an attestation step, giving it the keys necessary to access the encrypted checkpoint image. If the guest failed to complete these operations in a timely manner, the host could simply terminate it.

7.3 SGX optimisations

Besides the limitations identified in §5.4, two further opportunities exist to optimise SGX performance for Haven.

Exception handling As mentioned in §5.3 and §5.4, two aspects of SGX combine to substantially increase the overhead of exception handling: `ERESUME` and `IRET` are both illegal in an enclave. Haven's exception handler must `EEXIT` to a tiny stub that `ERESUME`s a modified context within the enclave. This then runs the LibOS and application exception handlers, which typically finish by executing `IRET` to restore the original context. However, this causes another illegal instruction exception. Overall, a single application exception (e.g., stack growth) results in two exceptions and eight enclave crossings. As there appear to be no insurmountable security implications, we suggest permitting (or providing equivalent replacements for) these instructions within the enclave.

Demand loading Haven's shield loads application and LibOS binaries. Modern systems typically load lazily: virtual address space is reserved, but pages are allocated and filled only on first access, in response to faults. Since other threads may also access the same pages while they are loaded, demand loading is done using a private memory mapping before remapping pages with appropriate permissions in the final location. However, since SGX

does not support moving an existing page, Haven must eagerly load all binaries. This adds time and memory overhead, particularly at startup. For example, running PowerShell until it displays a prompt causes 124MB of DLLs to be loaded, but only 4% of those pages are accessed.

The revised SGX specification includes an `EACCEPTCOPY` instruction [29], which allows a new page to be both allocated and initialised with a copy of data located elsewhere in the enclave before it becomes accessible to software. This should enable demand-loading, although we have not yet had the opportunity to experiment with an implementation.

7.4 Hardware for shielded execution

Shielded VMs As we noted in §2.1, SGX is the first commodity hardware that permits efficient multiplexing among multiple isolated programs without relying on trusted software. However, for many use-cases including cloud deployments, hardware capable of isolating full virtual machines (rather than portions of a user-mode address space, as in SGX) would be desirable from a compatibility standpoint: it would support complete guest operating systems. There are many performance and complexity-related challenges to building such hardware, including multiple levels of address translation, privileged instructions and virtual devices. However, if it were available, we suspect that a Haven-like shield module would be a suitable architecture to protect unmodified guest VMs from a malicious hypervisor, since the same trust issues addressed by Haven in the OS also arise in VM interfaces.

Shielding without information leakage Our definition of shielded execution (§2.1) requires confidentiality for intermediate state of the guest. As we noted (§3.1), SGX limits our ability to achieve this, because it exposes to the host information such as exceptions and page faults, and because side-channels such as cache footprint leak guest state information. At first glance, this concession seems necessary for the OS to dynamically manage resources. If all resources were allocated statically for the life of the guest, a host would have no reason to observe guest states. However, an OS relies on seeing application behaviour to efficiently multiplex resources over varying demands; e.g., by monitoring faulting addresses it can use page replacement algorithms to manage physical memory.

We conjecture that a hardware isolation mechanism supporting true shielded execution can in fact permit dynamic resource multiplexing by changing the role of the resource manager. Present mechanisms conflate determining the quantity of resources (e.g., the number of physical pages) to allocate with the selection of specific resources (the virtual-to-physical mapping). We propose decoupling these, giving the host control only over resource quantities, and allowing the guest to choose specific resources to relinquish when allocations change. For example, memory would be managed by allocating physical pages in the host, but allowing the guest to control its virtual mappings, and using self-paging [22] to permit oversubscription. The host may ask a guest to relinquish pages, and kill it if it did not meet a deadline. We anticipate that hardware could also support cache partitioning, achieving similar results to page colouring [64] without constraints on physical allocation; a host could flush and repartition caches without exposing guest access patterns.

8 Related Work

We survey related work in two areas: trusted hardware, and systems to isolate applications from an untrusted OS. Notwithstanding prior research [9, 12, 31, 34, 44], hardware security modules (HSMs) [1, 53], trusted platform modules (TPMs) [57] and ARM TrustZone [5] are presently the main hardware sources of trust on commodity platforms, and we focus on them.

Hardware security modules HSMs [53] are often used to protect high-value secrets (e.g., keys) in the cloud. An HSM is a protected computing element made tamper-proof using a physical barrier and a self-destruct mechanism to erase data if the barrier is compromised. Cloud HSMs such as AWS CloudHSM [1] offer APIs for key manipulation, signing, and encryption. As a result, the cloud user's keys are protected, but other data must still be transiently decrypted in a general-purpose node in order to use it. This reduces, but does not eliminate, the attack window compared to storing data persistently in the clear. As dedicated hardware, HSMs are also expensive.

Trusted hardware TPMs [57] are hardware devices included in many PCs supporting a similar attestation mechanism to SGX. The original approach to TPM-based attestation builds a chain of trust using progressive measurement of code during system boot, such as the bootloader, OS, etc. [50]. More recent CPU extensions enable the *late launch* and dynamic attestation of an isolated "secure kernel". This can reduce a platform's software TCB to just the late-launched code, a form of isolated execution, albeit one with two key drawbacks compared to SGX: vulnerability to relatively-simple hardware attacks including memory snooping, and lack of support for efficient multiplexing of distinct late-launch environments.

There are two general approaches to multiplexing TPM systems. The first, taken by Flicker [38], is to time-

multiplex the entire PC between secure kernels and an untrusted host OS. Unfortunately, because it uses a separate chip, TPM dynamic attestation is notoriously slow – Flicker's transition times are tens to hundreds of milliseconds for small modules. The second approach is to attest a trusted hypervisor or OS, which implements isolated execution in software [39, 49, 52]; the main downside for our scenario is that, regardless of its size, the hypervisor remains under the cloud provider's control. A cloud user may compare a TPM attestation to a known hash of the hypervisor binary, but we assume that the provider must be able to update the hypervisor (e.g., to patch security flaws, but also to insert arbitrary backdoors), and the user must ultimately trust them. This approach may be feasible given a hypervisor that is verified (down to binary code) to protect guest confidentiality and integrity, so that the attestation a user receives is meaningfully connected to a proof of the isolation mechanism, but current progress on OS-level formal verification is some way from this goal [23, 30].

A set of extensions in many ARM processors, TrustZone enables a "secure world" execution environment that is isolated from the OS [5]. Like the TPM, systems using TrustZone rely on software to multiplex the secure world; for example, to enable a runtime for security-critical components of mobile applications [51].

Shielding apps from an untrusted OS A number of systems seek to defend applications from a malicious OS. While XOMOS [35] used custom hardware, most recent approaches rely on the support of a trusted hypervisor. Proxos [54] runs isolated applications in a separate VM, but allows them to interact with a commodity OS. Overshadow [11] and SP^3 [60] pioneered transparent encryption of user memory when visible to the OS, protecting application data from direct tampering. CloudVisor [63] extended this technique to full VMs using nested virtualisation, while SecureME [12] accelerated it in hardware. More recently, InkTag [25] showed how to optimise the guest OS and protect persistent storage, and Virtual Ghost [14] used compiler techniques to implement a similar mechanism within the OS kernel.

However, systems based solely on protecting application memory from an untrusted OS are vulnerable to Iago attacks through the system call interface [10]. Systems such as InkTag [25] attempt to defeat Iago attacks by interposing on system calls (e.g., in a custom `libc`) and checking their results, but we feel that this approach is unlikely to be tractable for arbitrary applications given the complexity of modern OS interfaces – Linux today includes more than 300 system calls, and Windows well over 1000, as well as exceptions and asynchronous event mechanisms. Instead, Haven defeats Iago attacks by design, using a LibOS, shield module, and a substantially smaller (≈20 calls) mutually-distrusting host interface; it also avoids the need for a trusted hypervisor through SGX assistance.

Cloud security Finally, other research tackles the problem of removing trust from the cloud. Although fully homomorphic encryption schemes which allow arbitrary computation on encrypted data suffer intractably high overhead [20, 21], partially homomorphic encryption has been successfully applied in some domains; e.g., some database queries [4, 46] and MapReduce programs [55] can be implemented without ever decrypting data. While this cannot support existing applications, it also does not require trusted hardware.

MiniBox [33] combines the isolation of TrustVisor [39] with the sandbox of Native Client [61]. Like Haven, MiniBox achieves mutual distrust between application code and the host OS. Unlike Haven, MiniBox relies on a trusted hypervisor, and its isolated execution environment supports only small pieces of application logic, rather than complete unmodified applications.

PrivateCore vCage [48] is a virtual machine monitor implementing full memory encryption for commodity hardware by executing guest VMs entirely in-cache and encrypting their data before it is evicted to main memory. Although it relies on a trusted hypervisor, and thus cannot meet our security goals, it shares with SGX a resistance to memory probes and similar physical attacks.

9 Conclusion

Today's cloud platforms offer many advantages, but these are often outweighed by the risks inherent in a hierarchical security architecture: the provider is trusted with full access to user data. To eliminate this risk, Haven implements shielded execution of unmodified server applications in an untrusted cloud host. Haven brings us one step closer to a true "utility computing" model for the cloud, where the utility provides resources (processor cores, storage, and networking) but has no access to user data.

Acknowledgements

We appreciate the assistance and collaboration of Intel Labs, especially Matthew Hoekstra, Simon Johnson, Rebekah Leslie-Hurd, Frank McKeen, Carlos Rozas and Krystof Zmudzinski. We are also grateful to all who provided feedback, in particular Steve Hand, Jon Howell, Rebecca Isaacs, Rama Kotla, Bryan Parno, Oriana Riva, Emmett Witchel and the anonymous reviewers.

References

[1] *AWS CloudHSM Getting Started Guide*. Amazon Web Services, Nov. 2013. http://aws.amazon.com/cloudhsm/.

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems*, 10:53–79, 1992.

[4] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *6th Conference on Innovative Data Systems Research*, Jan. 2013.

[5] *Building a Secure System using TrustZone Technology*. ARM Limited, Apr. 2009. Ref. PRD29-GENC-009492C.

[6] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *EuroSys Conference*, pages 239–252, Apr. 2013.

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, pages 105–120, Aug. 2003.

[8] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Report 2006/052, Cryptology ePrint Archive, 2006.

[9] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *16th IEEE International Symposium on High-Performance Computer Architecture*, Jan. 2010.

[10] S. Checkoway and H. Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2013.

[11] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2008.

[12] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: a hardware-software approach to full system security. In *International Conference on Supercomputing*, pages 108–119, 2011.

[13] Cloud Security Alliance. Government access to information survey. https://cloudsecurityalliance.org/research/surveys/#_nsa_prism, July 2013.

[14] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting applications from hostile operating systems. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 81–96, 2014.

[15] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 339–354, Dec. 2008.

[16] K. Fu, F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *4th USENIX Symposium on Operating Systems Design and Implementation*, pages 181–196, 2000.

[17] B. Gassend, E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *9th IEEE International Symposium on High-Performance Computer Architecture*, pages 295–306, 2003.

[18] B. Gellman and L. Poitras. U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program. *The Washington Post*, June 2013.

[19] B. Gellman and A. Soltani. NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say. *The Washington Post*, Oct. 2013.

[20] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[21] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *32nd International Cryptology Conference*, 2012.

[22] S. M. Hand. Self-paging in the Nemesis operating system. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, 1999.

[23] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2014.

[24] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[25] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: secure applications on an untrusted operating system. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2013.

[26] F. Hou, N. Xiao, F. Liu, H. He, and D. Gu. Performance and consistency improvements of hash tree based disk storage protection. In *2009 IEEE International Conference on Networking, Architecture, and Storage (NAS 2009)*, pages 51–56, 2009.

[27] J. Howell, B. Parno, and J. R. Douceur. How to run POSIX apps in a minimal picoprocess. In *2013 USENIX Annual Technical Conference*, pages 321–332, June 2013.

[28] *Software Guard Extensions Programming Reference*. Intel

Corp., Sept. 2013. Ref. #329298-001 http://software.intel.com/sites/default/files/329298-001.pdf.

[29] *Software Guard Extensions Programming Reference, Rev. 2*. Intel Corp., Oct. 2014. Ref. #329298-002.

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles*, pages 207–220, 2009.

[31] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *32nd International Symposium on Computer Architecture*, pages 2–13, 2005.

[32] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in HYDRA. In *5th ACM Symposium on Operating Systems Principles*, pages 132–140, 1975.

[33] Y. Li, J. M. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. MiniBox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference*, June 2014.

[34] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[35] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *19th ACM Symposium on Operating Systems Principles*, pages 178–192, 2003.

[36] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *4th USENIX Symposium on Operating Systems Design and Implementation*, pages 135–150, 2000.

[37] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *13th ACM Symposium on Operating Systems Principles*, pages 110–121, Oct. 1991.

[38] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *EuroSys Conference*, pages 315–328, 2008.

[39] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, May 2010.

[40] D. McGrew and J. Viega. The Galois/counter mode of operation (GCM). http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf, 2004.

[41] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[42] R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology – CRYPTO'87*, pages 369–378, 1987.

[43] C. C. Miller. Revelations of N.S.A. spying cost U.S. tech companies. *The New York Times*, Mar. 2014.

[44] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *20th ACM Conference on Computer and Communications Security*, pages 13–24, 2013.

[45] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *IEEE Symposium on Security and Privacy*, pages 379–394, 2011.

[46] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *23rd ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[47] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinksy, and G. C. Hunt. Rethinking the library OS from the top down. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304, Mar. 2011.

[48] PrivateCore. Trustworthy computing for OpenStack with vCage. http://privatecore.com/vcage/, 2014.

[49] H. Raj, D. Robinson, T. B. Tariq, P. England, S. Saroiu, and A. Wolman. Credo: Trusted computing for guest VMs with a commodity hypervisor. Technical Report MSR-TR-2011-130, Microsoft Research, Dec. 2011.

[50] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *13th USENIX Security Symposium*, Aug. 2004.

[51] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–80, 2014.

[52] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *23rd ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[53] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(9):831–860, Apr. 1999. ISSN 1389-1286.

[54] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *7th USENIX Symposium on Operating Systems Design and Implementation*, pages 279–292, 2006.

[55] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein. MrCrypt: Static analysis for secure cloud computations. In *2013 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 271–286, 2013.

[56] *TPC benchmark E standard specification*. Transaction Processing Performance Council, June 2010. Rev. 1.12.0.

[57] *TPM Main Specification Level 2*. Trusted Computing Group, Mar. 2011. Version 1.2, Revision 116.

[58] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *EuroSys Conference*, Apr. 2014.

[59] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune. Trustworthy execution on mobile devices: What security properties can my mobile platform give *me*? In *5th International Conference on Trust and Trustworthy Computing*, pages 159–178, June 2012.

[60] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *4th International conference on Virtual Execution Environments*, pages 71–80, 2008.

[61] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[62] A. Yun, C. Shi, and Y. Kim. On protecting integrity and confidentiality of cryptographic file system for outsourced storage. In *2009 ACM Workshop on Cloud Computing Security*, pages 67–76, 2009.

[63] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *23rd ACM Symposium on Operating Systems Principles*, pages 203–216, 2011.

[64] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *EuroSys Conference*, pages 89–102, 2009.

Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing

Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou
Microsoft

Zhengping Qian, Ming Wu, Lidong Zhou
Microsoft Research

Abstract

Efficiently scheduling data-parallel computation jobs over cloud-scale computing clusters is critical for job performance, system throughput, and resource utilization. It is becoming even more challenging with growing cluster sizes and more complex workloads with diverse characteristics. This paper presents Apollo, a highly scalable and coordinated scheduling framework, which has been deployed on production clusters at Microsoft to schedule thousands of computations with millions of tasks efficiently and effectively on tens of thousands of machines daily. The framework performs scheduling decisions in a distributed manner, utilizing global cluster information via a loosely coordinated mechanism. Each scheduling decision considers future resource availability and optimizes various performance and system factors together in a single unified model. Apollo is robust, with means to cope with unexpected system dynamics, and can take advantage of idle system resources gracefully while supplying guaranteed resources when needed.

1 Introduction

MapReduce-like systems [7, 15] make data-parallel computations easy to program and allow running jobs that process terabytes of data on large clusters of commodity hardware. Each data-processing job consists of a number of tasks with inter-task dependencies that describe execution order. A task is a basic unit of computation that is scheduled to execute on a server.

Efficient scheduling, which tracks task dependencies and assigns tasks to servers for execution when ready, is critical to the overall system performance and service quality. The growing popularity and diversity of data-parallel computation makes scheduling increasingly challenging. For example, the production clusters that we use for data-parallel computations are growing in size, each with over 20,000 servers. A growing community of thousands of users from many different organizations submit jobs to the clusters every day, resulting in a peak rate of tens of thousands of scheduling requests per second. The submitted jobs are diverse in nature, with a variety of characteristics in terms of data volume to process, complexity of computation logic, degree of parallelism, and resource requirements. A scheduler must (i) scale to make tens of thousands of scheduling decisions per second on a cluster with tens of thousands of servers; (ii) maintain fair sharing of resources among different users and groups; and (iii) make high-quality scheduling decisions that take into account factors such as data locality, job characteristics, and server load, to minimize job latencies while utilizing the resources in a cluster fully.

This paper presents the Apollo scheduling framework, which has been fully deployed to schedule jobs in cloud-scale production clusters at Microsoft, serving a variety of on-line services. Scheduling billions of tasks daily efficiently and effectively, Apollo addresses the scheduling challenges in large-scale clusters with the following technical contributions.

- To balance scalability and scheduling quality, Apollo adopts a *distributed* and (loosely) *coordinated* scheduling framework, in which independent scheduling decisions are made in an optimistic and coordinated manner by incorporating synchronized cluster utilization information. Such a design strikes the right balance: it avoids the suboptimal (and often conflicting) decisions by independent schedulers of a completely decentralized architecture, while removing the scalability bottleneck and single point of failure of a centralized design.

- To achieve high-quality scheduling decisions, Apollo schedules each task on a server that minimizes the task completion time. The estimation model incorporates a variety of factors and allows a scheduler to perform a weighted decision, rather than solely considering data locality or server load. The data parallel nature of computation al-

lows Apollo to refine the estimates of task execution time continuously based on observed runtime statistics from similar tasks during job execution.

- To supply individual schedulers with cluster information, Apollo introduces a lightweight hardware-independent mechanism to advertise load on servers. When combined with a local task queue on each server, the mechanism provides a near-future view of resource availability on all the servers, which is used by the schedulers in decision making.

- To cope with unexpected cluster dynamics, suboptimal estimations, and other abnormal runtime behaviors, which are facts of life in large-scale clusters, Apollo is made robust through a series of *correction mechanisms* that dynamically adjust and rectify suboptimal decisions at runtime. We present a unique *deferred correction mechanism* that allows resolving conflicts between independent schedulers only if they have a significant impact, and show that such an approach works well in practice.

- To drive high cluster utilization while maintaining low job latencies, Apollo introduces *opportunistic scheduling*, which effectively creates two classes of tasks: *regular tasks* and *opportunistic tasks*. Apollo ensures low latency for regular tasks, while using the opportunistic tasks for high utilization to fill in the slack left by regular tasks. Apollo further uses a *token* based mechanism to manage capacity and to avoid overloading the system by limiting the total number of regular tasks.

- To ensure no service disruption or performance regression when we roll out Apollo to replace a previous scheduler deployed in production, we designed Apollo to support *staged rollout* to production clusters and *validation at scale*. Those constraints have received little attention in research, but are nevertheless crucial in practice and we share our experiences in achieving those demanding goals.

We observe that Apollo schedules over 20,000 tasks per second in a production cluster with over 20,000 machines. It also delivers high scheduling quality, with 95% of regular tasks experiencing a queuing delay of under 1 second, while achieving consistently high (over 80%) and balanced CPU utilization across the cluster.

The rest of the paper is organized as follows. Section 2 presents a high-level overview of our distributed computing infrastructure and the query workload that Apollo supports. Section 3 presents an architectural overview, explains the coordinated scheduling in detail, and describes the correction mechanisms. We describe our engineering experiences in developing and deploying Apollo to our cloud infrastructure in Section 4. A thorough evaluation is presented in Section 5. We review related work in Section 6 and conclude in Section 7.

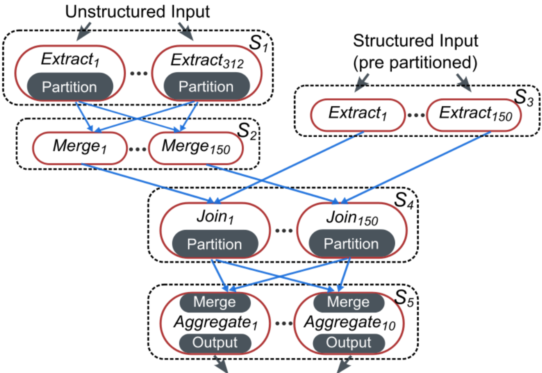


Figure 1: A sample SCOPE execution graph.

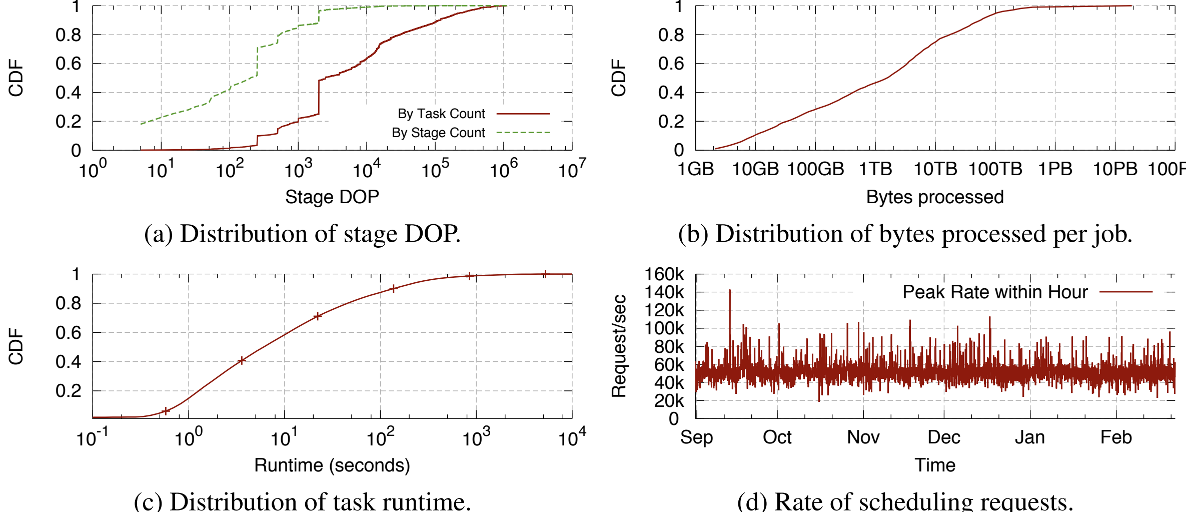
2 Scheduling at Production Scale

Apollo serves as the underlying scheduling framework for Microsoft's distributed computation platform, which supports large-scale data analysis for a variety of business needs. A typical cluster contains tens of thousands of commodity servers, interconnected by an oversubscribed network. A distributed file system stores data in *partitions* that are distributed and replicated, similar to GFS [12] and HDFS [3]. All computation jobs are written using SCOPE [32], a SQL-like high-level scripting language, augmented with user-defined processing logic. The optimizer transforms a job into a physical execution plan represented as a directed acyclic graph (DAG), with tasks, each representing a basic computation unit, as vertices and the data flows between tasks as edges. Tasks that perform the same computation on different partitions of the same inputs are logically grouped together in *stages*. The number of tasks per stage indicates the degree of parallelism (DOP).

Figure 1 shows a sample execution graph in SCOPE, greatly simplified from an important production job that collects user click information and derives insights for advertisement effectiveness. Conceptually, the job performs a join between an unstructured user log and a structured input that is pre-partitioned by the join key. The plan first partitions the unstructured input using the partitioning scheme from the other input: stages S_1 and S_2 respectively partition the data and aggregate each partition. A partitioned join is then performed in stage S_4. The DOP is set to 312 for S_1 based on the input data volume, set to 10 for S_5, and set to 150 for S_2, S_3, and S_4.

2.1 Capacity Management and Tokens

In order to ensure fairness and predictability of performance, the system uses a *token-based* mechanism to allocate capacity to jobs. Each token is defined as the right



(a) Distribution of stage DOP.

(b) Distribution of bytes processed per job.

(c) Distribution of task runtime.

(d) Rate of scheduling requests.

Figure 2: Heterogeneous workload.

to execute a regular task, consuming up to a predefined amount of CPU and memory, on a machine in the cluster. For example, if a job has an allocation of 100 tokens, this means it can run 100 tasks, each of which consumes up to a predefined maximum amount of CPU and memory.

A *virtual cluster* is created for each user group for security and resource sharing reasons. Each virtual cluster is assigned a certain amount of capacity in terms of number of tokens, and maintains a queue of all submitted jobs. A job submission contains the target virtual cluster, the necessary credentials, and the required number of tokens for execution. Virtual cluster management utilizes various admission control policies and decides how and when to assign its allocated tokens to submitted jobs. Jobs that do not get their required tokens will be queued in the virtual cluster. The system also supports a wide range of capabilities, such as job priorities, suspension, upgrades, and cancellations.

Once a job starts to execute with required tokens, it is a scheduler's responsibility to execute its optimized execution plan by assigning tasks to servers while respecting token allocation, enforcing task dependencies, and providing fault tolerance.

2.2 The Essence of Job Scheduling

Scheduling a job involves the following responsibilities: (i) *ready list*: maintain a list of tasks that are ready to be scheduled: initially, the list includes those leaf tasks that operate on the original inputs (e.g., tasks in stages S_1 and S_3 in Figure 1); (ii) *task priority:* sort the ready list appropriately; (iii) *capacity management:* manage the capacity assigned to the job and decide when to schedule a task based on the capacity management policy; (iv) *task scheduling:* decide where to schedule a task and dispatch it to the selected server; (v) *failure recovery:* monitor scheduled tasks, initiate recovery actions when tasks fail, and mark the job failed if recovery is not possible; (vi) *task completion:* when a task completes, check its dependent tasks in the execution graph and move them to the *ready list* if all the tasks that they depend on have completed; (vii) *job completion:* repeat the whole process until all tasks in the job are completed.

2.3 Production Workload Characteristics

The characteristics of our target production workloads greatly influence the Apollo design. Our computation clusters run more than 100,000 jobs on a daily basis. At any point in time, there are hundreds of jobs running concurrently. Those jobs vary drastically in almost every dimension, to meet a wide range of business scenarios and requirements. For example, large jobs process terabytes to petabytes of data, contain sophisticated business logic with a few dozen complex joins, aggregations, and user-defined functions, have hundreds of stages, contain over a million tasks in the execution plan, and may take hours to finish. On the other hand, small jobs process gigabytes of data and can finish in seconds. In SCOPE, different jobs are also assigned with different amounts of resources. The workload evolves constantly as the supporting business changes over time. This workload diversity poses tremendous challenges for the underlying scheduling framework to deal with efficiently and effectively. We describe several job characteristics in our production environment to illustrate the diverse and dynamic nature of the computation workload.

In SCOPE, the DOP for a stage in a job is chosen based on the amount of data to process and the complexity of each computation. Even within a single job, the DOP changes for different stages as the data volume changes over the job's lifetime. Figure 2(a) shows the distribution

of stage DOP in our production environment. It varies from a single digit to a few tens of thousands. Almost 40% of stages have a DOP of less than 100, accounting for less than 2% of the total workload. More than 98% of tasks are part of stages with DOP of more than 100. These large stage sizes allow a scheduler to draw statistics from some tasks to infer behavior of other tasks in the same stage, which Apollo leverages to make informed and better decisions. Job sizes vary widely from a single vertex to millions of vertices per job graph. As illustrated in Figure 2(b), the amount of data processed per job ranges from gigabytes to tens of petabytes. Task execution times range from less than 100ms to a few hours, as shown in Figure 2(c). 50% of tasks run for less than 10 seconds and are sensitive to scheduling latency. Some tasks require external files such as executables, configurations, and lookup tables for their execution, thus incurring initialization costs. In some cases, such external files required for execution are bigger than the actual input to be processed, which means locality should be based on where those files are cached rather than input location. Collectively, such a large number of jobs create a high scheduling-request rate, with peaks above 100,000 requests per second, as shown in Figure 2(d).

The very dynamic and diverse characteristics of our computing workloads and cluster environments impose several challenges for the scheduling framework, including scalability, efficiency, robustness, and resource usage balance. The Apollo scheduling framework has been designed and shown to address these challenges over large production clusters at Microsoft.

3 The Apollo Framework

To support the scale and scheduling rate required for the production workload, Apollo adopts a *distributed* and *coordinated* architecture, where the scheduling of each job is performed independently and incorporates aggregated global cluster load information.

3.1 Architectural Overview

Figure 3 provides an overview of Apollo's architecture. A *Job Manager* (JM), also called a scheduler, is assigned to manage the life cycle of each job. The global cluster load information used by each JM is provided through the cooperation of two additional entities in the Apollo framework: a *Resource Monitor* (RM) for each cluster and a *Process Node* (PN) on each server. A PN process running on each server is responsible for managing the local resources on that server and performing local scheduling, while the RM aggregates load information from PNs across the cluster continuously, providing a global view of the cluster status for each JM to make informed scheduling decisions.

While treated as a single logical entity, the RM can be implemented physically in different configurations with different mechanisms, as it essentially addresses the well-studied problem of monitoring dynamically changing state of a collection of distributed resources at a large scale. For example, it can use a tree hierarchy [20] or a directory service with an eventually consistent gossip protocol [8, 26]. Apollo's architecture can accommodate any of such configurations. We implemented the RM in a master-slave configuration using Paxos [18]. The RM is never on the performance critical path: Apollo can continue to make scheduling decisions (at a degraded quality) even when the RM is temporarily unavailable, for example, during a transient master-slave switch due to a machine failure. In addition, once a task is scheduled to a PN, the JM obtains up-to-date load information directly from the PN via frequent status updates.

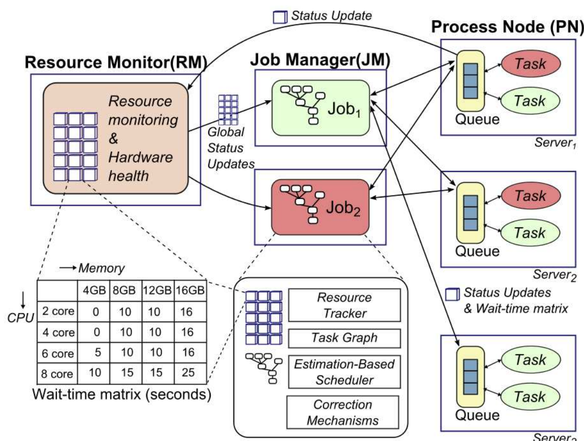


Figure 3: Apollo architectural overview.

To better predict resource utilization in the near future and to optimize scheduling quality, each PN maintains a local queue of tasks assigned to the server and advertises its future resource availability in the form of a *wait-time matrix* inferred from the queue (Section 3.2). Apollo thereby adopts an estimation-based approach to making task scheduling decisions. Specifically, Apollo considers the wait-time matrices, aggregated by the RM, together with the individual characteristics of tasks to be scheduled, such as the location of inputs (Section 3.3). However, cluster dynamics pose many challenges in practice; for example, the wait-time matrices might be stale, estimates might be suboptimal, and the cluster environment might sometimes be unpredictable. Apollo therefore incorporates correction mechanisms for robustness and dynamically adjusts scheduling decisions at runtime (Section 3.4). Finally, there is an inherent tension between providing guaranteed resources to jobs (e.g., to ensure SLAs) and achieving high cluster utilization, because both the load on a cluster and the resource needs of a job fluctuate constantly. Apollo resolves this tension through *opportunistic scheduling*, which creates second-class tasks to use idle resources (Section 3.5).

3.2 PN Queue and Wait-Time Matrix

The PN on each server manages a *queue* of tasks assigned to the server in order to provide projections on future resource availability. When a JM schedules a task on a server, it sends a task-creation request with (i) fine grained resource requirement (CPU cores and memory), (ii) estimated runtime, and (iii) a list of files required to run the task (e.g., executables and configuration files). Once a task creation request is received, the PN copies the required files to a local directory using a peer-to-peer data transfer framework combined with a local cache. The PN monitors CPU and memory usage, considers the resource requirements of tasks in the queue, and executes them when the capacity is available. It maximizes resource utilization by executing as many tasks as possible, subject to the CPU and memory requirements of individual tasks. The PN queue is mostly FIFO, but can be reordered. For example, a later task requiring a smaller amount of resources can fill a gap without affecting the expected start time of others.

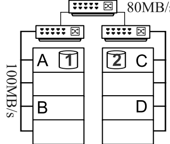
The use of task queues enables schedulers to dispatch tasks to the PNs proactively based on future resource availability, instead of based on instantaneous availability. As illustrated later in Section 3.3, Apollo considers task wait time (for sufficient resources to be available) and other task characteristics holistically to optimize task scheduling. The use of task queues also masks task initialization cost by copying the files before execution capacity is available, thereby avoiding idle gaps between tasks. Such a direct-dispatch mechanism provides the efficiency needed particularly by small tasks, for which any protocol to negotiate incurs significant overhead.

The PN also provides feedback to the JM to help improve accuracy of task runtime estimation. Initially, the JM uses conservative estimates provided by the query optimizer [32] based on the operators in a task and the amount of data to be processed. Tasks in the same stage perform the same computation over different datasets. Their runtime characteristics are similar and the statistics from the executions of the earlier tasks can help improve runtime estimates for the later ones. Once a task starts running, the PN monitors its overall resource usage and responds to the corresponding JM's status update requests with information such as memory usage, CPU time, execution time (wall clock time), and I/O throughput. The JM then uses this information along with other factors such as operator characteristics and input size to refine resource usage and predict expected runtime for tasks from the same stage.

The PN further exposes the load on the current server to be aggregated by its RM. Its representation of the load information should ideally convey a projection of the future resource availability, mask the heterogeneity of servers in our data centers (e.g., servers with 64GB of memory and 128GB of memory have different capacities), and be concise enough to allow frequent updates. Apollo's solution is a *wait-time matrix*, with each cell corresponding to the expected wait time for a task that requires a certain amount of CPU and memory. Figure 3 contains a matrix example: the value 10 in cell ⟨12 GB, 4 cores⟩ denotes that a task that needs 4 CPU cores and 12GB of memory has to wait 10 seconds in this PN before it can get its resource quota to execute. The PN maintains a matrix of expected wait times for any hypothetical future task with various resource quotas, based on the currently running and queued tasks. The algorithm simulates local task execution and evaluates how long a future task with a given CPU/memory requirement would wait on this PN to be executed. The PN updates this matrix frequently by considering the actual resource situation and the latest task runtime and resource estimates. Finally, the PN sends this matrix, along with a timestamp, to every JM that has running or queued tasks in this PN. It also sends the matrix to the RM using a heartbeat mechanism.

3.3 Estimation-Based Scheduling

A JM has to decide which server to schedule a particular task to using the wait-time matrices in the aggregated view provided by the RM and the individual characteristics of the task to be scheduled. Apollo has to consider a variety of (often conflicting) factors that affect the quality of scheduling decisions and does so in a single unified model using an estimation-based approach.



(a) Server map.

Server	Wait	I/O	Wait+I/O
A	0s	63.13s	63.13s
B	0s	63.5s	63.5s
C	40s	32.50s	72.50s
D	5s	51.25s	56.25s

(b) Scheduling alternatives.

Figure 4: A task scheduling example.

We use an example to illustrate the importance of considering various factors all together, as well as the benefit of having a local queue on each server. Figure 4(a) shows a simplified server map with two racks, each with four servers, connected via a hierarchically structured network. Assume data can be read from local disks at 160MB/s, from within the same rack at 100MB/s, and from a different rack at 80MB/s. Consider scheduling a task with two inputs (one 100MB stored on server A and the other 5GB stored on server C) whose runtime is dominated by I/O. Figure 4(b) shows the four scheduling choices, where servers A and B are immediately available, while server C has the best data locality. Yet, D is the optimal choice among those four choices. This can be recognized only when we consider data locality

and wait time together. This example also illustrates the value of local queues: without a local queue on each server, any scheduling mechanism that checks for immediate resource availability would settle on the non-optimal choice of server A or B.

Apollo therefore considers various factors holistically and performs scheduling by estimating task completion time. First, we estimate the task completion time if there is no failure, denoted by E_{succ}, using the formula

$$E_{succ} = I + W + R \tag{1}$$

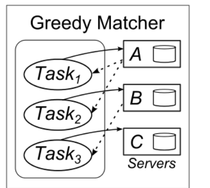
I denotes the initialization time for fetching the needed files for the task, which could be 0 if those files are cached locally. The expected wait time, denoted as W, comes from a lookup in the wait-time matrix of the target server with the task resource requirement. The task runtime, denoted as R, consists of both I/O time and CPU time. The I/O time is computed as the input size divided by the expected I/O throughput. The I/O could be from local memory, disks, or network at various bandwidths. Overall, estimation of R initially incorporates information from the optimizer and is refined with runtime statistics from the tasks in the same stage.

Second, we consider the probability of task failure to calculate the final completion time estimate, denoted by C. Hardware failures, maintenance, repairs, and software deployments are inevitable in a real large-scale environment. To mitigate their impact, the RM also gathers information on upcoming and past maintenance scheduled on every server. Together, a success probability P_{succ} is derived and considered to calculate C, as shown below. A penalty constant K_{fail}, determined empirically, is used to model the cost of server failure on the completion time.

$$C = P_{succ} E_{succ} + K_{fail}(1 - P_{succ}) E_{succ} \tag{2}$$

Task Priorities. Besides completion time estimation, the task-execution order also matters for overall job latency. For example, for the job graph in Figure 1, the tasks in S_1 run for 1 minute on average, the tasks in S_2 run for an average of 2 minutes, with potential partition-skew induced stragglers running up to 10 minutes, and the tasks in S_3 run for 30 seconds on average. As a result, efficiently executing S_1 and S_2 surely appears more critical to achieve the fastest runtime. Therefore, the scheduler should prioritize resources to S_1 and S_2 before considering S_3. Within S_2, the scheduler should start the vertex with the largest input as early as possible, because it is the most likely to be on the critical path of the job.

A static task priority is annotated per stage by the optimizer through analyzing the job DAG and calculating the potential critical path of the job execution. Tasks within a stage are prioritized based on the input size. Apollo schedules tasks and allocates their resources in a descending order of their priorities. Since a job contains a finite number of tasks, the starvation of a task with low static priority is impossible, because eventually it will be the only task left to execute, and will be executed.



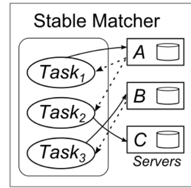


Figure 5: A matching example.

Stable Matching. For efficiency, Apollo schedules tasks with similar priorities in *batches* and turns the problem of task scheduling into that of *matching* between tasks and servers. For each task, we could search all the servers in a cluster for the best match. The approach becomes prohibitively expensive on a large cluster. Instead, Apollo limits the search space for a task to a *candidate set* of servers, including (i) a set of servers on which inputs of significant sizes are located (ii) a set of servers in the same rack as those from the first group (iii) two servers randomly picked from a set of lightly-loaded servers; the list is curated in the background.

A greedy algorithm can be applied for each task *sequentially*, choosing the server with the earliest estimated completion time at each step. However, the outcome of the greedy algorithm is sensitive to the order in which tasks are matched and often leads to suboptimal decisions. Figure 5 shows an example with a batch of three tasks being scheduled. Assume both $Task_1$ and $Task_2$ read data from server A while $Task_3$ reads from server B, as shown with dotted lines. Each server has capacity to start one task. The greedy matcher first matches $Task_1$ to server A, then matches $Task_2$ to server B because $Task_1$ is already scheduled on A, and finally $Task_3$ to server C, as shown with solid lines. A better match would have assigned $Task_3$ to server B for better locality.

Apollo therefore adopts a variant of the stable matching algorithm [10] to match tasks with servers. For each task in a batch, Apollo finds the server with the earliest estimated completion time as a *proposal* for that task. A server accepts a proposal from a task if that is the only proposal assigned. A conflict arises when more than one task proposes to the same server. In this case, the server picks the task whose completion time saving is the greatest if it is assigned to the server. The tasks not picked withdraw their proposals and enter the next iteration that tries to match the remaining tasks and servers. The algorithm iterates until all tasks have been assigned, or until it reaches the maximum number of iterations. As shown in Figure 5, the stable matcher matches $Task_2$ to C and $Task_3$ to B, which effectively leverages locality and results in better job performance.

The scheduler then sorts all the matched pairs based on their *quality* to decide the dispatch order. A match is considered with a higher quality if its task has a lower server wait time. The scheduler iterates over the sorted matches and dispatches in order until it is out of the allocated capacity. If opportunistic scheduling (Section 3.5) is enabled, the scheduler continues to dispatch the tasks until the opportunistic scheduling limit.

To simplify the matching algorithm for a tradeoff between efficiency and quality, Apollo assigns only one task to each server in a single batch, because otherwise Apollo has to update the wait-time matrix for a server to take into account the newly assigned task, which increases the complexity of the algorithm. This simplification might lead to a suboptimal match for a task in a case where servers taking on a task in the same batch already remains a better choice. Apollo mitigates the effect in two ways: if the suboptimal match is of a low quality, sorting the matches by quality will cause the dispatching of this task to be postponed, and later re-evaluated. Even if the suboptimal match is dispatched, the correction mechanisms described in Section 3.4 are designed to catch this case and reschedule the task if needed.

3.4 Correction Mechanisms

In Apollo, each JM can schedule tasks independently at a high frequency, with no delay in the process. This is critical for scheduling a large number of small tasks in the workload. However, due to the distributed nature of the scheduling, several JMs might make competing decisions at the same time. In addition, the information used, such as wait-time matrices, for scheduling decisions might be stale; the task wait time and runtime might be under or overestimated. Apollo has built-in mechanisms to address those challenges and dynamically adjust scheduling decisions with new information.

Unlike previous proposals (e.g., as in Omega [23]) in which conflicts are immediately handled at the *scheduling time*, Apollo optimistically *defers* any correction until after tasks are dispatched to PN queues. This design choice is based on our observation that conflicts are not always harmful. Two tasks scheduled to the same server simultaneously by different job managers might be able to run concurrently if there are sufficient resources for both; tasks that are previously scheduled on the server might complete soon, releasing the resources early enough to make any conflict resolution unnecessary. In those cases, a deferred correction mechanism, made possible with local queues, avoids the unnecessary overhead associated with eager detection and resolution. Correction mechanisms continuously re-evaluate the scheduling decisions with up-to-date information and make appropriate adjustments whenever necessary.

Duplicate Scheduling. When a JM gets fresh information from a PN during task creation, task upgrade, or while monitoring its queued tasks, it compares the information from this PN (and the elapsed wait time so far) to the information that was used to make the scheduling decision. The scheduler re-evaluates the decision if (i) the updated expected wait time is significantly higher than the original; (ii) the expected wait time is greater than the average among the tasks in the same stage; (iii) the elapsed wait time is already greater than the average. The first condition indicates an underestimated task completion time on the server, while the second/third conditions indicate a low matching quality. Any change in the decision triggers scheduling a duplicate task to a new desired server. Duplicates are discarded when one task starts.

Randomization. Multiple JMs might schedule tasks to the same lightly loaded PN, not aware of each other, thereby leading to scheduling *conflicts*. Apollo adds a small random number to each completion time estimation. This random factor helps reduce the chances of conflicts by having different JMs choose different, almost equally desirable, servers. The number is typically proportional to the communication interval between the JM and the PN, introducing no noticeable impact on the quality of the scheduling decisions.

Confidence. The aggregated cluster information obtained from the RM contains wait-time matrices of different ages, some of which can be stale. The scheduler attributes a lower confidence to older wait-time matrices because it is likely that the wait time changed since the time the matrix was calculated. When the confidence in the wait-time matrix is low, the scheduler will produce a pessimistic estimate by looking up the wait time of a task consuming more CPU and memory.

Straggler Detection. Stragglers are tasks making progress at a slower rate than other tasks, and have a crippling impact on job performances [4]. Apollo's straggler detection mechanism monitors the rate at which data is processed and the rate at which CPU is consumed to predict the amount of time remaining for each task. Other tasks in the same stage are used as a baseline for comparison. When the time it would take to rerun a task is significantly less than the time it would take to let it complete, a duplicate copy is started. They will execute in parallel until the first one finishes, or until the duplicate copy caught up with the original task. The scheduler also monitors the rate of I/O and detects stragglers caused by slow intermediate inputs. When a task is slow because of abnormal I/O latencies, it can rerun a copy of the upstream task to provide an alternate I/O path.

3.5 Opportunistic Scheduling

Besides achieving high quality scheduling at scale, Apollo is also designed to operate efficiently and drive high cluster utilization. Cluster utilization fluctuates over

time for several reasons. First, not all users submit jobs at the same time to consume their allocated capacities fully. A typical example is that the cluster load on weekdays is always higher than on weekends. Second, jobs differ in their resource requirements. Even daily jobs with the same computation logic consume different amount of resources as their input data sizes vary. Finally, a complete job typically goes through multiple stages, with different levels of parallelism and varied resource requirements. Such load fluctuation on the system provides schedulers with an opportunity to improve job performance by increasing utilization, at the cost of predictability. How to judiciously utilize occasionally idle computation resources *without* affecting SLAs remains challenging.

We introduce *opportunistic scheduling* in Apollo to gracefully take advantage of idle resources whenever they are available. Tasks can execute either in the *regular* mode, with sufficient tokens to cover its resource consumption, or in the *opportunistic* mode, without allocated resources. Each scheduler first applies optimistic scheduling to dispatch regular tasks with its allocated tokens. If all the tokens are utilized and there are still pending tasks to be scheduled, opportunistic scheduling may be applied to dispatch opportunistic tasks. Performance degradation of regular task is prevented by running opportunistic tasks at a lower priority at each server, and any opportunistic task can be preempted or terminated if the server is under resource pressure.

One immediate challenge is to prevent one job from consuming all the idle resources unfairly. Apollo uses *randomized allocation* to achieve probabilistic resource fairness for opportunistic tasks. In addition, Apollo upgrades opportunistic tasks to regular ones when tokens become available and assigned.

Randomized Allocation Mechanism. Ideally, the opportunistic resources should be shared fairly among jobs, proportionally to jobs' token allocation. This is particularly challenging as both the overall cluster load and individual server load fluctuate over time, which makes it difficult, if not impossible, to guarantee absolute instantaneous fairness. Instead, we focus on avoiding the worst case of a few jobs consuming all the available capacity of the cluster and target average fairness.

Apollo achieves this by setting a maximum opportunistic allowance for a given job proportionally to its token allocation. For example, a job with n tokens can have up to cn opportunistic tasks dispatched for some constant c. When a PN has spare capacity and the regular queue is empty, the PN picks a *random* task to execute from the opportunistic-task queue, regardless of when it was dispatched. If the chosen task requires more resources than what is available, the randomized selection process continues until there is no more task that can execute. Compared to a FIFO queue, the algorithm has the benefit of allowing jobs that start later to get a share of the capacity quickly. If a FIFO queue were used for opportunistic tasks, it could take an arbitrary amount of time for a later task to make its way through the queue, offering unfair advantages to tasks that start earlier.

As the degree of parallelism for a job varies in its lifetime, the number of tasks that are ready to be scheduled also varies. As a result, a job may not always be able to dispatch enough opportunistic tasks to use its opportunistic allowance fully. We further enhance the system by allowing each scheduler to increase the weight of an opportunistic task during random selection, to compensate for the reduction in the number of tasks. For example, a weight of 2 means a task has twice the probability to be picked. The total weight of all opportunistic tasks issued by the job must not exceed its opportunistic allowance.

Under an ideal workload, in which tasks run for the same amount of time and consume the same amount of resources, and in a perfectly balanced cluster, this strategy averages to sharing the opportunistic resources proportionally to the job allocation. However, in reality, tasks have large variations in runtime and resource requirements. The number of tasks dispatched per jobs change constantly as tasks complete and new tasks become ready. Further, jobs may not have enough parallelism at all times to use their opportunistic allowance fully. Designing a fully *decentralized* mechanism that maintains a strong fairness guarantee in a dynamic environment remains a challenging topic for future work.

Task Upgrade. Opportunistic tasks are subject to starvation if the host server experiences resource pressure. Further, the opportunistic tasks can wait for an unbounded amount of time in the queue. In order to avoid job starvation, tasks scheduled opportunistically can be upgraded to regular tasks after being assigned a token. Because a job requires at least one token to run and there is a finite amount of tasks in a job, the scheduler is able to transition a starving opportunistic task to a regular task at one point, thus preventing job starvation.

After an opportunistic task is dispatched, the scheduler tracks the task in its ready list until it completes. When scheduling a regular task, the scheduler considers both unscheduled tasks and previously scheduled opportunistic tasks that still wait for execution. Each scheduler allocates its tokens to tasks and performs task matches in a descending order of their priorities. It is not required that an opportunistic task be upgraded on the same machine, but it might be preferable as there is no initialization time. By calculating all costs holistically, the scheduler favors upgrading opportunistic tasks on machines with fewer regular tasks, while waiting for temporarily heavily loaded machines to drain. This strategy results in a better utilization of the tokens and better load balancing.

4 Engineering Experiences

Before the development of Apollo, we already had a production system running at full scale with our previous generation DAG scheduler, also referred to as the baseline scheduler. The baseline scheduler schedules jobs without any global cluster load information. When tasks wait in a PN queue, the baseline scheduler systematically triggers duplicates to explore other idle PNs and balance the load. Locality is modeled as a per-task scheduling constraint that is relaxed over time, instead of using completion time estimation.

In our first attempt to improve the baseline scheduler, we took an approach of a centralized global scheduler, which is responsible for scheduling all jobs in a cluster. While the global scheduler theoretically oversees all activities in the cluster and could make optimal scheduling decisions, we found that it became a performance bottleneck and its scheduling quality degraded as the numbers of machines in the cluster and concurrent jobs continued to grow. In addition, as described in Section 2, our heterogeneous workload consists of jobs with diverse characteristics. Any scheduling delay could have a direct and severe impact on small tasks. Such lessons ultimately led us to choose Apollo's distributed and loosely coordinated scheduling paradigm.

During the development of Apollo, we had to ensure the availability of our production system throughout the process, from early prototyping and experimentation to evaluation and eventual full deployment. This has posed many interesting challenges in validation, migration, deployment, and operation at full production scale.

Validation at Scale. We started by evaluating Apollo in an isolated test cluster at a smaller scale. It helps us verify scheduling decisions at every step and track down performance issues quickly. However, the approach has limitations as the scale of the test cluster is rather limited and cannot emulate the dynamic cluster environment. Many interesting challenges arise as the scale and complexity of the workload grows. For example, Apollo and the baseline scheduler make different assumptions around the capacity of the machines. In a test cluster with a single job running at a time, the baseline scheduler schedules a single task to a server, resulting in much lower machine utilization compared to Apollo. However, this improvement does not translate into gain in production environments because the utilization in production clusters is already high. Therefore, it is important for us to evaluate Apollo in real production clusters, side by side with busy production workloads.

Another lesson we learned from our first failed attempt was to validate design and performance continuously, instead of delaying full validation until completion and exposing scalability and performance issues when it is too late. This is particularly important as each engineering attempt is significant and time-consuming at this large scale.

Apollo's fully decentralized design allows each scheduler to run side by side with other schedulers, or other versions of Apollo itself. Such engineering agility is critical and allows us to compare performance across different schedulers at scale in the same production environments. We sampled production jobs and reran them twice, one with the baseline scheduler and the other with the Apollo scheduler, to compare the job performance. In order to minimize other random factors, we initially ran them side by side. However, the approach resulted in artificial resource contention as both jobs read the same inputs. Instead, we chose to run the two jobs one after another. Our experiences show that the cluster load is unlikely to change drastically between the two consecutive runs. We also modified the baseline scheduler to produce accurate estimates for task runtime and resource requirements using Apollo's logic so that Apollo could perform adequately well in this mixed mode environment. This allowed us to get performance data from early exposure to large scale environment in the design and experimentation phase.

Migration at Scale. We designed Apollo to replace the previous scheduler *in place*. On the one hand, this means that protocols had to be carefully designed to be compatible; on the other hand, it also means that we had to make sure both schedulers could coexist in the same environment, each scheduling a part of the workload on the same set of machines, without creating interferences. For example, Apollo judiciously performs opportunistic scheduling with significantly less resource. In isolation, Apollo issues 98% less duplicates than the baseline scheduler, without losing performance. However, in the mixed mode where both schedulers runs, the baseline scheduler gains an unfair advantage by issuing more duplicates to get work done. We therefore tuned the system during the transition to increase the probability to start opportunistic tasks for Apollo-scheduled jobs in order to correct the bias caused by the reduction in the number of tasks scheduled.

Deployment at Scale. Without any service downtime or unavailability, we rolled out Apollo to our users in *stages* and increased user coverage over time until eventually fully deployed on all clusters. At each stage, we closely watched various system metrics, verified job performance, and studied impact on other system components before proceeding to the next stage.

One interesting observation was that users who had not yet migrated to Apollo also experienced some performance improvement during the deployment process. This was because Apollo avoided scheduling tasks to hotspots inside the system, which helped improve job performance across the cluster, including the ones sched-

uled by the baseline scheduler. When we finally enabled those jobs with Apollo, the absolute percentage of improvement was less than what we observed initially.

Operation at Scale. Even with thoughtful design and implementation, the dynamic behavior of such a large scale system continues to pose new challenges, which motivate us to refine and improve the system continuously while operating Apollo at scale. For example, Apollo leverages an opportunistic scheduling mechanism to increase system utilization by operating tasks either in normal-priority *regular* mode or in lower-priority *opportunistic* mode. Initially, this priority is enforced by local operating system but not by the underlying distributed file system. During the early validation, this did not pose a problem. However, as we deployed Apollo, both CPU and I/O utilization in the cluster increased. The overall increase in I/O pressure caused a latency impact and interfered with tasks even running in *regular* mode. This highlights the importance of interpreting task priority throughout the entire stack to maintain predictable performance at high utilization.

Another example is that we developed a server failure model by mining historical data and various factors to predict the likelihood of possible repeated server failures in the near future, based on recent observed events. As described in Section 3.3, such failure model has a direct impact on task completion time estimation and thus influences scheduling decisions. The model works great in most cases and helps Apollo avoid scheduling tasks to repeated offenders, thereby improving system reliability. However, a rare power transformer outage caused failures on a large number of servers all at once. After the power was restored, the model predicted that they were likely to fail again and prevented Apollo from using those machines. As a result, the recovery of the cluster was unnecessarily slowed down. This indicates the importance of further distinguishing failure types and dealing with them presumably in different models.

5 Evaluation

Since late 2013, Apollo has been deployed in production clusters at Microsoft, each containing over 20,000 commodity servers. To evaluate Apollo thoroughly, we use a combination of the following methods: (i) We analyze and report various system metrics on large-scale production clusters where Apollo has been deployed; further, we compare the behavior before and after enabling Apollo. (ii) We perform in-depth studies on representative production jobs to highlight our observations at per-job level. (iii) We use trace-driven simulations on certain specific points in the Apollo design to compare with alternatives. Whenever possible, we prefer reporting the production Apollo behavior, instead of using simulated results, because it is hard, if not infeasible, to model the complexity of real workload and production environment faithfully in a simulator. To our knowledge, this is the first detailed analysis of production schedulers at such a large scale with such a complex and diverse workload.

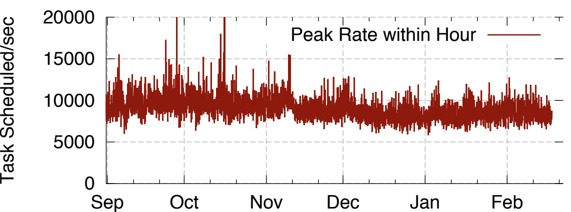


Figure 6: Scheduling rates.

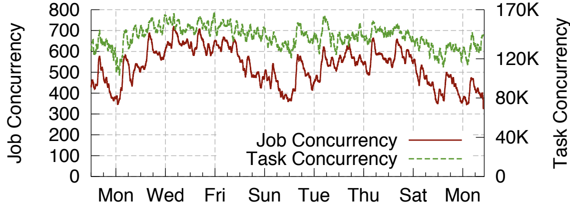
We intend to answer the following questions: (a) How well does Apollo scale and utilize resources in a large-scale cluster? (b) What is the scheduling quality with Apollo? (c) How accurate are the estimates on task execution and queuing time used in Apollo and how much do the estimates help? (d) How does Apollo cope with dynamic cluster environment? (e) What is the complexity of Apollo's core scheduling algorithm?

5.1 Apollo at Scale

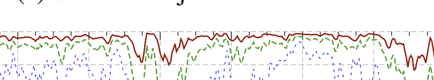
We first measured the aggregated scheduling rate in a cluster over time to understand Apollo's scalability. We define *scheduling rate* as the number of *materialized* scheduling decisions (those resulting a task execution at a PN) made by all the individual schedulers in the cluster per second. Figure 6 shows the peak *scheduling rates* for each hour over the past 6 months, highlighting that Apollo can constantly provide a scheduling rate of above 10,000, reaching up to 20,000 per second in a single cluster. This confirms the need for a distributed scheduling infrastructure, as it would be challenging for any single scheduler to make high quality decisions at this rate. It is important to note that the *scheduling rate* is also governed by the capacity of the cluster and the number of concurrent jobs. With its distributed architecture, we expect the scheduling rate to increase with the number of jobs and the size of the cluster.

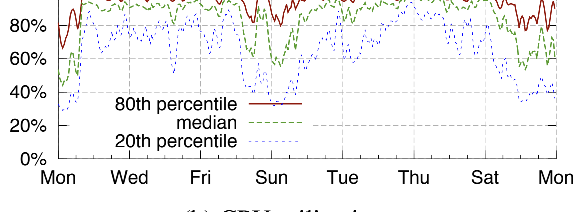
We then drill down into a period of two weeks and report various aspects of Apollo, without diluting data over a long period of time. Figure 7(a) shows the number of concurrently running jobs and their tasks in the cluster while Figure 7(b) shows server CPU utilization in the same period, both sampled at every 10 seconds. Apollo is able to run 750 concurrent complex jobs (140,000 concurrent regular tasks) and achieve over 90% CPU utilization when the demand is high during the weekdays, reaching closely the capacity of the cluster.

To illustrate the distribution of system utilization among all the servers in the cluster, Figure 7(b) shows the median, as well as the 20th and 80th percentiles in CPU utilization. When the demand surges, Apollo makes use of all the available resources and only leaves a 3%

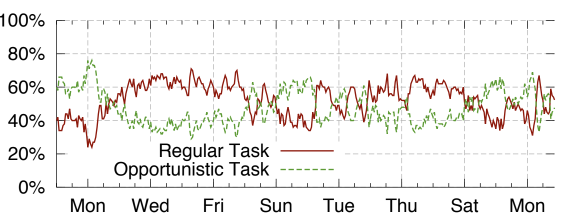


(a) Concurrent jobs and tasks.





(b) CPU utilization.



(c) CPU time breakdown: regular and opportunistic task.

Figure 7: Apollo in production.

gap between the 20th and the 80th percentiles. When the demand is low, such as on Sundays, the load is less balanced and the machine utilization is mostly correlated to the popularity of the data stored on them. The figures shows a clear weekly pattern of the workloads. The task concurrency decreases during the weekends and recovers back to a large volume during the weekdays. However, the dip in system utilization is significantly less than that of the number of jobs submitted. With opportunistic scheduling, Apollo allows jobs to gracefully exploit idle system resources to achieve better job performances and continues to drive system utilization high even with fewer number of jobs. This is further validated in Figure 7(c), which shows the percentage of CPU hours attributed to regular and opportunistic tasks. During the weekdays, 70% of Apollo's workload comes from regular tasks. The balance shifts during the weekends: more opportunistic tasks get executed on the available resources when there are fewer regular tasks.

Task location	% Tasks	% I/Os
The same server that contains the input	28%	46%
Within the same rack as the input	56%	47%
Across rack	16%	7%

Table 1: Breakdown of tasks and their I/Os.

We also measured the average task queuing time for all regular tasks to verify that the queuing time remains low despite the high concurrency and system utilization. At the 95th percentile, the tasks show less than 1 second queuing time across the entire cluster. Apollo achieves this by considering data locality, wait time, and other factors holistically when distributing tasks. Table 1 categorizes tasks into three groups and reports the percentage of I/Os they account for. 72% of the tasks are dispatched to servers that require reading inputs remotely, either within or across rack, to avoid wait time. If only data locality is considered, tasks are likely to concentrate on a small group of servers that contain hot data.

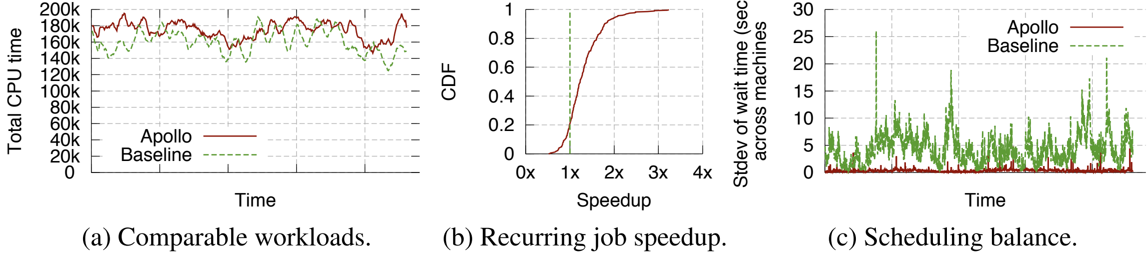
Summary. Combined, those results show that Apollo is highly scalable, capable of scheduling over 20,000 requests per second, and driving high and balanced system utilization while incurring minimum syqueuing time.

5.2 Scheduling Quality

We evaluate the scheduling quality of Apollo in two ways: (i) compare with the previously implemented baseline scheduler using production workloads and (ii) study business critical production jobs and use trace-based simulations to compare the quality.

Performing a fair comparison between the baseline scheduler and Apollo in a truly production environment with real workload is challenging. Fortunately, we replaced the baseline scheduler in place with Apollo, allowing us to observe both schedulers in the same cluster with similar workloads. Further, about 40% of the production jobs in the cluster has a *recurring* pattern and such recurring jobs account for more than 75% system resource utilization [5]. We therefore choose two time frames, before and after the Apollo deployment, to compare performance and speedup of each recurring job, running Apollo and the baseline scheduler respectively. The recurring nature of the workload produced a strong correlation in CPU time between the workloads in the two time frames, as shown in Figure 8(a). Figure 8(b) shows the CDF of the speedups for all recurring jobs and it indicates that about 80% of recurring jobs receive various degrees of performance improvements (up to 3x in speedup) with Apollo. To understand the reason for the differences, we measured the average task queuing time on each server for every window of 10 minutes. Figure 8(c) shows the standard deviation of the average task queue time across servers, comparing Apollo with the baseline scheduler, which indicates clearly that Apollo achieves much more balanced task queues across servers.

For the second experiment, we present a study of one business critical production job, which runs every hour. The job consumes logs from a search and advertisement engine and analyzes user click information. Its execution graph consists of around ten thousands tasks, processing a few terabytes of data. The execution graph shown in Figure 1 is a much simplified version of this



(a) Comparable workloads. (b) Recurring job speedup. (c) Scheduling balance.

Figure 8: Comparison between Apollo and the baseline scheduler.

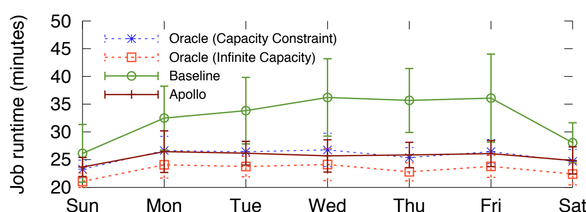


Figure 9: Job latencies with different schedulers.

job. The performance of the job varies by weekdays because of periodic fluctuations in the input volume of user clicks. To compare performance, we use one job per day at the same hour in a week and evaluated the scheduling performance of Apollo, baseline scheduler, and a simulated oracle scheduler, which has zero task wait time, zero scheduling latency, zero task failures, and knows exact runtime statistics about each task. Further, we use two variants of the oracle scheduler: (i) oracle with capacity constraint, which is limited to the same capacity that was allocated to the job when it ran in the production environment and (ii) oracle without capacity constraint, which has access to unlimited capacity, roughly representing the best case scenario.

Figure 9 shows job performance using Apollo and the baseline scheduler, respectively, and compares them with the oracle scheduler using runtime traces. On average, the job latency improved around 22% with Apollo over the baseline scheduler, and Apollo performed within 4.5% of the oracle scheduler. On some days, Apollo is even better than the oracle scheduler with the capacity constraint because the job is able to get some extra speedup from opportunistic scheduling, allowing the job to get more capacity than the capacity constraint used by the oracle scheduler.

Summary. Apollo delivers excellent job performance compared with the baseline scheduler and its scheduling quality is close to the optimal case.

5.3 Evaluating Estimates

Estimating task completion time, as described in Section 3.3, plays an important role in Apollo's scheduling algorithm and thus job performance. Both task initialization time and I/O time can be calculated when the inputs and server locations are known at runtime.

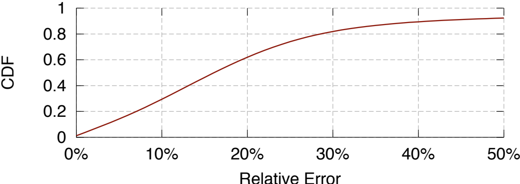


Figure 10: CPU time estimation.

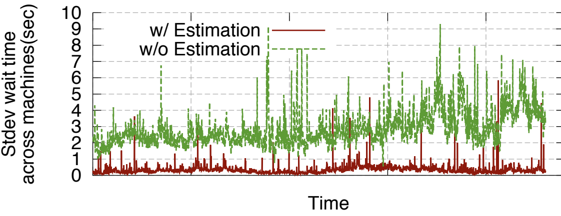


Figure 11: Scheduling balance (estimation effect).

We first measure the accuracy of the estimated task wait time, as a result of applying task resource estimation to the wait-time matrix. Over 95% tasks have a wait time estimation error of less than 1 second. We then measure the CDF of the estimation error for task CPU time, as shown in Figure 10. For 75% of tasks, the CPU time predicted when the task is scheduled is within 25% of the actual CPU consumption. Apollo continues to refine runtime estimates based on statistics from the finished tasks within the same stage at runtime. Nevertheless, a number of factors make runtime estimation challenging. A common case is for tasks with *early-out* behavior without reading all of its input. An example of such tasks may consist of a filter operator followed by a `TOP` N operator. Different tasks may consume different amount of input data before collecting N rows satisfying the filter condition, which makes inference based on past task runtime difficult. Complex user code whose resource consumption and execution time varies by input data characteristics also makes prediction difficult, if not infeasible. We evaluate how Apollo dynamically adjusts scheduling decisions at runtime in Section 5.4.

In order to evaluate the overall estimation impact, we compare the Apollo performance with and without estimation. As we rolled out Apollo to one production cluster, we went through a phase in which we used a default estimate for all tasks uniformly, before we enabled all the internal estimation mechanism. We refer the phase

as Apollo without estimation. Comparing the system behavior before and after allows us to understand the impact of estimation on scheduling decisions in a production cluster because the workloads are similar as we reported in Section 5.2. Figure 11 shows the distributions of task queuing time. With estimation enabled, Apollo achieves much more balanced scheduling across servers, which in turn leads to shorter task queuing latency.

Summary. Apollo provides good estimates on task wait time and CPU time, despite all the challenges, and estimation does help improve scheduling quality. Further improvements can be achieved by leveraging statistics of recurring jobs and better understanding task internals, which is part of our future work.

5.4 Correction Effectiveness

In case of inaccurate estimates or sudden changes in a cluster environment, Apollo applies a series of correction mechanisms to mitigate effectively. For duplicate scheduling, we call a duplicate task successful if it starts before the initial task.

Conditions (W: wait time)	Trigger rate	Success rate
New expected W significantly higher	0.12%	81.3%
Expected W greater than average	0.12%	81.3%
Elapsed W greater than average	0.17%	83.0%

Table 2: Duplicate scheduling efficiency.

Table 2 evaluates different heuristics of duplicate scheduling, described in Section 3.4, for the same two-week period and reports how frequently they are triggered and their success rate. Overall, Apollo's duplicate scheduling is efficient, with 82% success rates, and accounts for less than 0.5% of task creations. Such a low correction rate confirms the viability of optimistic scheduling and deferred corrections for this workload.

Straggler detection and mitigation is also important for job performance. Apollo is able to catch more than 70% stragglers efficiently and apply mitigation timely to expedite query execution. We omit the detailed experiments due to space constraints.

Summary. Apollo's correction mechanisms are shown effective with small overhead.

5.5 Stable Matching Efficiency

In a general case, the complexity of the stable matching algorithm, when using a red-black tree to maintain a sorted set of tasks to schedule, is $O(n^2)$ while the greedy algorithm is $O(n \log(n))$. However in our case the complexity of the stable matching algorithm is limited to $O(n \log(n))$. The algorithm usually converges in less than 3 iterations and our implementation limits the number of iterations of the matcher, which makes the worst case complexity $O(n \log(n))$, the same as the greedy algorithm. In practice, a scheduling batch contains less than 1,000 tasks and the computation overhead is negligible, with no observed performance differences.

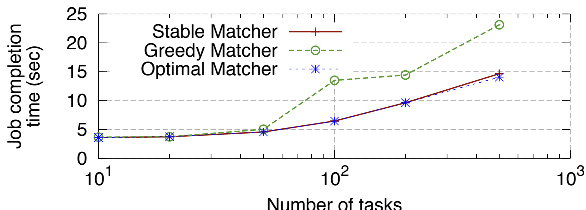


Figure 12: Matching quality.

We verify the effectiveness of the stable matching algorithm using a simulator. We measure the amount of time it takes for a workload of N tasks to complete on 100 servers, using the greedy, stable matching, and optimal matching algorithms, respectively. The optimal matching algorithm uses an exhaustive search to compute the best possible sequence of scheduling. Each server has a single execution slot and have an expected wait time that is exponentially distributed with an average of 1. The expected runtime of each tasks is exponentially distributed with an average of 1. Each task is randomly assigned a server preference and runs faster on the preferred server. Figure 12 shows that the stable matching algorithm performs within 5% of the optimal matching under the simulated conditions while the greedy approach, which schedules tasks one at a time on the server with the minimum expected completion time, was 57% slower than the optimal matching.

Summary. Apollo's matching algorithm has the same asymptotic complexity as a naive greedy algorithm with negligible overhead. It performs significantly better than the greedy algorithm and is within 5% of the optimal scheduling in our simulation.

6 Related Work

Job scheduling was extensively studied [24, 25] in high-performance computing for scheduling batch CPU-intensive jobs and has become a hot topic again with emerging data-parallel systems [7, 15, 29]. Monolithic schedulers, such as Hadoop Fair Scheduler [28] and Quincy [16], implement a scheduling policy for an entire cluster using a centralized component. This class of schedulers suffers from scalability challenges when serving large-scale clusters.

Mesos [14] and YARN [27] aim to ease the support for multiple workloads by decoupling resource management from application logic in a two-layer design. Facebook Corona [1] uses a similar design but focuses on reducing job latency and improving scalability for Hadoop by leveraging a push-based message flow and an optimistic locking pattern. Mesos, YARN, and Corona remain fundamentally centralized. Apollo in contrast makes decentralized scheduling decisions with no central scheduler, which facilitates small task scheduling.

Distributed schedulers such as Omega [23] and Sparrow [22] address the scalability challenges by not relying on a centralized entity for scheduling. Two different approaches have been explored to resolve conflicts. Omega resolves any conflict using optimistic concurrency control [17] where only one of the conflicting schedulers succeeds and the others have to roll back and retry later. Sparrow instead relies on (random) sampling and speculative scheduling to balance the load.

Similar to Omega, schedulers in Apollo makes optimistic scheduling decisions based on their views of the cluster (with the help of the RM). Unlike Omega, which detects and resolves conflicts at the scheduling time, Apollo is optimistic in conflict detection and resolution by deferring any corrections until after tasks are dispatched. This is made possible by Apollo's design of having local queues on servers. The use of local task queue and task runtime estimates provides critical insight about future resource availability and allows Apollo schedulers to optimize task scheduling by estimating task completion time, instead of based on instantaneous resource availability at the scheduling time. This also gives Apollo the extra benefit of masking resource-prefetching latency effectively, which is important for our target workload.

Although both adopting a distributed scheduling framework, Sparrow and Apollo differ in how they make scheduling decisions. Sparrow's sampling mechanism will schedule tasks on machines with the shortest queue, with no consideration for other factors affecting the completion time, such as locality. In contrast, Apollo schedulers optimize task scheduling by estimating task completion time that takes into account multiple factors such as load and locality. Sparrow uses reservation on multiple servers with late binding to alleviate its reliance on queue length, rather than task completion time. Such a reservation mechanism would introduce excessive initialization costs on multiple servers in our workload. Apollo introduces duplicate scheduling only as a correction mechanism; it is rarely triggered in our system.

While Omega and Sparrow have been evaluated using simulation and a 110-machine cluster respectively, our work distinguishes itself by showing Apollo's effectiveness in a real production environment at a truly large scale, with diverse workloads, and complex resource and job requirements.

Capacity management often goes hand-in-hand with scheduling. Capacity scheduler [2] in Hadoop/YARN uses global capacity queues to specify the share of resources for each job, which is similar to token-based resource guarantee implemented in Apollo. Apollo uses fine grained allocations and opportunistic scheduling to take advantage of idle resources gracefully. Resource management on local servers is also critical. Existing work leverages Linux containers [13], usage monitoring [27] and/or contention detection [31] to provide performance isolation. Apollo can accommodate any of those mechanisms.

Operator runtime estimation based on data statistics and operator semantics has been extensively studied in the database community for effective query optimization [19, 11, 6]. For distributed computing of arbitrary input and program, most effort (e.g., ParaTimer [21]) has been focusing on estimating the progress of running jobs based on runtime statistics. Apollo combines both static and runtime information and leverages program patterns (e.g., stage) to estimate task runtime. Apollo also includes a set of mechanisms to compensate inaccuracy whenever needed. For example, many existing works on outlier detection and straggler mitigation (e.g., LATE [30], Mantri [4], and Jockey [9]) are complementary to our work and can be integrated with the Apollo framework for reducing job latency.

7 Conclusion

In this paper, we present Apollo, a scalable and coordinated scheduling framework for cloud-scale computing. Apollo adopts a distributed and loosely coordinated scheduling architecture that scales well without sacrificing scheduling quality. Each Apollo scheduler considers various factors holistically and performs estimation-based scheduling to minimize task completion time. By maintaining a local task queue on each server, Apollo enables each scheduler to reason about future resource availability and implement a deferred correction mechanism to effectively adjust suboptimal decisions dynamically. To leverage idle system resources gracefully, opportunistic scheduling is used to maximize the overall system utilization. Apollo has been deployed on production clusters at Microsoft: it has been shown to achieve high utilization and low latency, while coping well with the dynamics in diverse workloads and large clusters.

Acknowledgements

We are grateful to our shepherd Andrew Warfield for his guidance in the revision process and to the anonymous reviewers for their insightful comments. David Chu and Jacob Lorch provided feedback that helped improve the paper. We would also like to thank the members of Microsoft SCOPE team for their contributions to the SCOPE distributed computation system, of which Apollo serves as the scheduling component. The following people contributed to our scheduling framework: Haochuan Fan, Jay Finger, Sapna Jain, Bikas Saha, Sergey Shelukhin, Sen Yang, Pavel Yatsuk, and Hongbo Zeng. Finally, we would like to thank Microsoft Big Data team members for their support and collaboration.

References

[1] Under the hood: Scheduling MapReduce jobs more efficiently with Corona. `http://on.fb.me/TxUsYN`, 2012. [Online; accessed 16-April-2014].

[2] Hadoop Capacity Scheduler. `http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`, 2014. [Online; accessed 16-April-2014].

[3] Hadoop Distributed File System (HDFS). `http://hadoop.apache.org/`, 2014. [Online; accessed 16-April-2014].

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in MapReduce clusters using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[5] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, and J. Zhou. Recurring job optimization in scope. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 805–806, New York, NY, USA, 2012. ACM.

[6] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[9] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[10] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–15, Jan. 1962.

[11] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang. Query optimization in the IBM DB2 family. *IEEE Data Eng. Bull.*, 16(4):4–18, 1993.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[13] M. Helsley. LXC: Linux container tools. *IBM devloperWorks Technical Library*, 2009.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[16] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[17] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[18] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[19] L. F. Mackert and G. M. Lohman. *R* optimizer validation and performance evaluation for local queries*, volume 15. ACM, 1986.

[20] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[21] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A progress indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 507–518. ACM, 2010.

[22] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[23] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

[24] G. Staples. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

[25] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[26] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2):164–206, 2003.

[27] V. K. Vavilapalli. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. SOCC*, 2013.

[28] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[30] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

[31] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM.

[32] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel databases meet MapReduce. *VLDB J.*, 21(5):611–636, 2012.

The Power of Choice in Data-Aware Cluster Scheduling

Shivaram Venkataraman[1], Aurojit Panda[1], Ganesh Ananthanarayanan[2], Michael J. Franklin[1], Ion Stoica[1]
[1]*UC Berkeley* [2]*Microsoft Research*

Abstract

Providing *timely* results in the face of rapid growth in data volumes has become important for analytical frameworks. For this reason, frameworks increasingly operate on only a *subset* of the input data. A key property of such sampling is that combinatorially many subsets of the input are present. We present KMN, a system that leverages these choices to perform *data-aware* scheduling, i.e., minimize time taken by tasks to read their inputs, for a DAG of tasks. KMN not only uses choices to co-locate tasks with their data but also percolates such combinatorial choices to downstream tasks in the DAG by launching a *few additional* tasks at every upstream stage. Evaluations using workloads from Facebook and Conviva on a 100-machine EC2 cluster show that KMN reduces average job duration by 81% using just 5% additional resources.

1 Introduction

Data-intensive computation frameworks drive many modern services like web search indexing and recommendation systems. Computation frameworks (e.g., Hadoop [14], Spark [60], Dryad [38]) translate a *job* into a DAG of many small *tasks*, and execute them efficiently on compute *slots* across large clusters. Tasks of *input* stages (e.g., map in MapReduce or extract in Dryad) read their data from distributed storage and pass their outputs to the downstream *intermediate* tasks (e.g., reduce in MapReduce or full-aggregate in Dryad).

The efficient execution of these predominantly I/O-intensive tasks is predicated on *data-aware scheduling*, i.e., minimizing the time taken by tasks to read their data. Widely deployed techniques for data-aware scheduling execute tasks on the same machine as their data (if the data is on one machine, as for input tasks) [8, 59] and avoid congested network links (when data is spread across machines, as for intermediate tasks) [13, 24]. However, despite these techniques, we see that production jobs in Facebook's Hadoop cluster are slower by 87% compared to *perfect data-aware scheduling* (§2.3). This is because, in multi-tenant clusters, compute slots that are ideal for data-aware task execution are often unavailable.

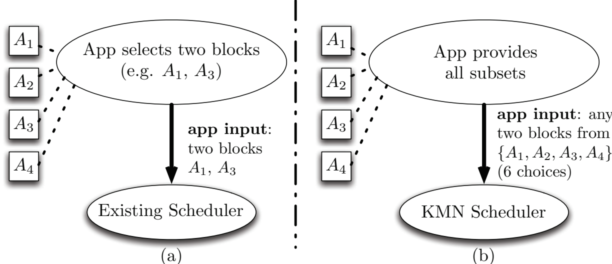


Figure 1: **"Late binding" allows applications to specify more inputs than tasks and schedulers dynamically choose task inputs at execution time.**

The importance of data-aware scheduling is increasing with rapid growth in data volumes [31]. To cope with this data growth and yet provide timely results, there is a trend of jobs using only a *subset* of their data. Examples include sampling-based approximate query processing systems [5,12] and machine learning algorithms [16,42]. A key property of such jobs is that they can compute on *any of the combinatorially many* subsets of the input dataset without compromising application correctness. For example, say a machine learning algorithm like stochastic gradient descent [16] needs to compute on a 5% uniform random sample of data. If the data is spread over 100 blocks then the scheduler can choose any 5 blocks and has $\binom{100}{5}$ input choices for this job.

Our goal is to *leverage the combinatorial choice of inputs for data-aware scheduling*. Current schedulers require the application to select a subset of the data on which the scheduler runs the job. This prevents the scheduler from taking advantage of available choices. In contrast, we argue for "late binding" *i.e.,* choosing the subset of data dynamically depending on the current state of the cluster (see Figure 1). This dramatically increases the number of data local slots for input tasks (e.g., map tasks), which increases the probability of achieving data locality even during high cluster utilizations.

Extending the benefits of choice to intermediate stages (e.g., reduce) is challenging because they consume all the outputs produced by upstream tasks. Thus, they have no choice in picking their inputs. When upstream outputs are not evenly spread across machines, the over-subscribed network links, typically cross-rack switch

links [24], become the bottleneck. We introduce choices for intermediate stages by launching a *small number of additional tasks* in the previous stage. As an example, consider a job with 400 map tasks and 50 reduce tasks. By launching 5% extra map tasks (420 tasks), we can pick the 400 map outputs that best avoid congested links.

Choosing the best upstream outputs used by intermediate stages is non-trivial due to complex communication patterns like many-to-many (for reduce tasks) and many-to-one (for joins). In fact, selecting the best upstream outputs can be shown to be NP-hard. We develop an efficient round-robin heuristic that attempts to balance data transfers evenly across cross-rack switch links. Further, upstream tasks do not all finish simultaneously due to *stragglers* [13, 61]. We handle stragglers in upstream tasks using a *delay*-based approach that balances the gains in balanced network transfers against the time spent waiting for stragglers. In the above example, for instance, it may schedule reduce tasks based on the earliest 415 map tasks and ignore the last 5 stragglers.

In summary, we make the following contributions:

- Identify the trend of combinatorial choices in inputs of jobs and leverage this for data-aware scheduling.
- Extend the benefits of choices to a DAG of stages by running a *few* extra tasks in each upstream stage.
- Build KMN, a system for analytics frameworks to seamlessly benefit from the combinatorial choices.

We have implemented KMN inside Spark [60]. We evaluate KMN using jobs from a production workload at Conviva, a video analytics company, and by replaying a trace from a production Hadoop cluster at Facebook. Our experiments on an EC2 cluster with 100 machines show that we can reduce average job duration by 81% (93% of ideal improvements) compared to Spark's scheduler. Our gains are due to KMN achieving 98% memory locality for input tasks and improving intermediate data transfers by 48%, while using $\leq$ 5% *extra resources*.

2 Choices and Data-Awareness

In this section we first discuss application trends that result in increased choices for scheduling (§2.1). We then explain data-aware scheduling (§2.2) and quantify its potential benefit in production clusters (§2.3).

2.1 Application Trends

With the rapid increase in the volume of data collected, it has become prohibitively expensive for data analytics frameworks to operate on all of the data. To provide timely results, there is a trend towards trading off accuracy for performance. Quick results obtained from just part of the dataset are often *good enough*.

(1) Approximate Query Processing: Many analytics frameworks support approximate query processing (AQP) using standard SQL syntax (e.g., BlinkDB [5], Presto [29]). They power many popular applications like exploratory data analysis [19, 54] and interactive debugging [3]. For example, products analysts could use AQP systems to quickly decide if an advertising campaign needs to be changed based on a sample of click through rates. AQP systems can bound both the time taken and the quality of the result by selecting appropriately sized inputs (samples) to met the deadline and error bound. Sample sizes are typically small relative to the original data (often, one-twentieth to one-fifth [43]) and many equivalent samples exist. Thus, sample selection presents a significant opportunity for smart scheduling.

(2) Machine Learning: The last few years has seen the deployment of large-scale distributed machine learning algorithms for commercial applications like spam classification [40] and machine translation [18]. Recent advances [17] have introduced stochastic versions of these algorithms, for example stochastic gradient descent [16] or stochastic L-BFGS [53], that can use *small random data samples* and provide statistically valid results even for large datasets. These algorithms are iterative and each iteration processes only a small sample of the data. Stochastic algorithms are agnostic to the sample selected in each iteration and support flexible scheduling.

(3) Erasure Coded Storage: Rise in data volumes have also led to clusters employing efficient storage techniques like erasure codes [50]. Erasure codes provide fault tolerance by storing k extra parity blocks for every n data blocks. Using *any* n data blocks of the $(n+k)$ blocks, applications can compute their input. Such storage systems also provide choices for data-aware scheduling.

Note that while the above applications provide an opportunity to pick *any* subset of the input data, our system can also handle custom sampling functions, which generate samples based on application requirements.

2.2 Data-Aware Scheduling

Data aware scheduling is important for both the input as well as intermediate stages of jobs due to their IO-intensive nature. In the input stage, tasks reads their input from a single machine and the natural goal is *locality* i.e. to schedule the task on a machine that stores its input (§2.2.1). For intermediate stages, tasks have their input spread across multiple machines. In this case, it is not possible to co-locate the task with all its inputs. Instead, the goal in this case is to schedule the task at a machine

that minimizes the time it takes to transfer all remote inputs. As over-subscribed cross-rack links are the main bottleneck in reads [22], we seek to *balance* the utilization of these links (§2.2.2).

2.2.1 Memory Locality for Input Tasks

Riding on the trend of falling memory prices, clusters are increasingly caching data in memory [11, 58]. As memory bandwidths are about $10\times-100\times$ greater than the fastest network bandwidths, data reads from memory provide dramatic acceleration for the IO-intensive analytics jobs. However, to reap the benefits of in-memory caches, tasks have to be scheduled with *memory locality*, i.e., on the same machine that contains their input data. Obtaining memory locality is important for timely completion of interactive approximate queries [9]. Iterative machine learning algorithms typically run 100's of iterations and lack of memory locality results in huge slowdown per iteration and the overall job.

Achieving memory locality is a challenging problem in clusters. Since in-memory storage is used only as a cache, data stored in memory is typically not replicated. Further, the amount of memory in a cluster is relatively small (often by three orders of magnitude [9]) when compared to stable storage: this difference means that replicating data in memory is not practical. Therefore, techniques for improving locality [8] developed for disk-based replicated storage are insufficient; they rely on the probability of locality increasing with the number of replicas. Further, as job completion times are dictated by the slowest task in the job, improving performance requires memory locality for *all* its tasks [11].

These challenges are reflected in production Hadoop clusters. A Facebook trace from 2010 [8, 21] shows that less than 60% of tasks achieve locality *even with* three replicas. As in-memory data is not replicated, it is harder for jobs to achieve memory locality for all their tasks.

2.2.2 Balanced Network for Intermediate Tasks

Intermediate stages of a job have communication patterns that result in their tasks reading inputs from many machines (e.g., all-to-all "shuffle" or many-to-one "join" stages). For I/O intensive intermediate tasks, the time to access data across the network dominates the running time, more so when intermediate outputs are stored in memory. Despite fast network links [56] and newer topologies [6, 35], bandwidths between machines connected to the same rack switch are still $2\times$ to $5\times$ higher than to machines outside the rack switch via the network core. Thus the runtime for an intermediate stage is dictated by the amount of data transferred across racks. Prior work has also shown that reducing cross-rack hotspots, i.e., optimizing the *bottleneck* cross-rack link [13, 24] can significantly improve performance.

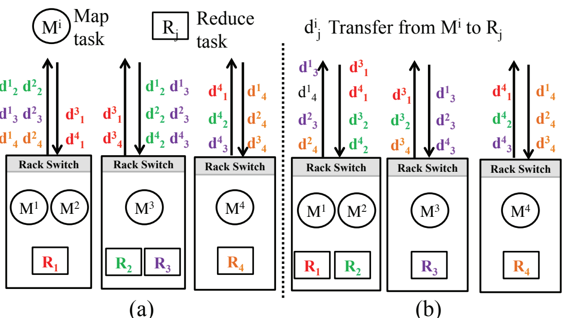


Figure 2: **Value of balanced network usage for a job with 4 map tasks and 4 reduce tasks. The left-hand side has unbalanced cross-rack links (maximum of 6 transfers, minimum of 2) while the right-hand side has better balance (maximum of 4 transfers, minimum of 3).**

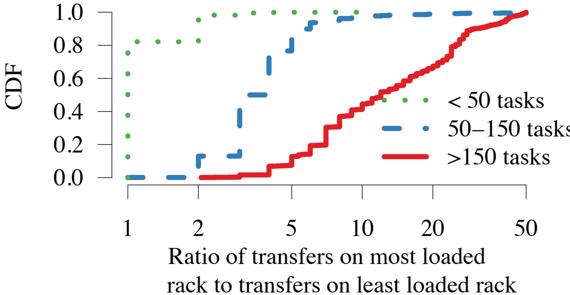


Figure 3: **CDF of cross-rack skew for the Facebook trace split by number of map tasks. Reducing cross-rack skew improves intermediate stage performance.**

Given the over-subscribed cross-rack links and the slowest tasks dictating job completion, it is important to *balance* traffic on the cross-rack links [15]. Figure 2 illustrates the result of having unbalanced cross-rack links. The schedule in Figure 2(b) results in a *cross-rack skew*, i.e., ratio of the highest to lowest used network links, of only $\frac{4}{3}$ (or 1.33) as opposed to $\frac{6}{2}$ (or 3) in Figure 2(a).

To highlight the importance of cross-rack skew, we used a trace of Hadoop jobs run in a Facebook cluster from 2010 [21] and computed the cross-rack skew ratio. Figure 3 shows a CDF of this ratio and is broken down by the number of map tasks in the job. From the figure we can see that for jobs with $50-150$ map tasks more than half of the jobs have a cross-rack skew of over $4\times$. For larger jobs we see that the median is $15\times$ and the 90th percentile value is in excess of $30\times$.

2.3 Potential Benefits

How much do the above-mentioned lack of locality and imbalanced network usage hurt jobs? We estimate the potential for data-aware scheduling to speed up jobs using the same Facebook trace (described in detail in §6).

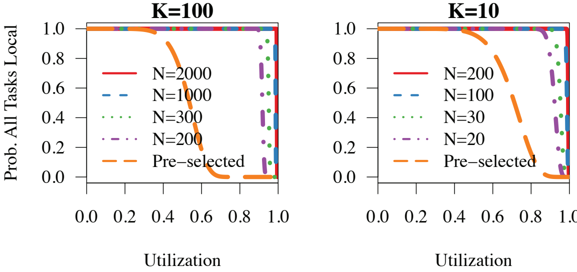


Figure 4: **Probability of input-stage locality when choosing any K out of N blocks. The scheduler can choose to execute a job on any of $\binom{K}{N}$ samples.**

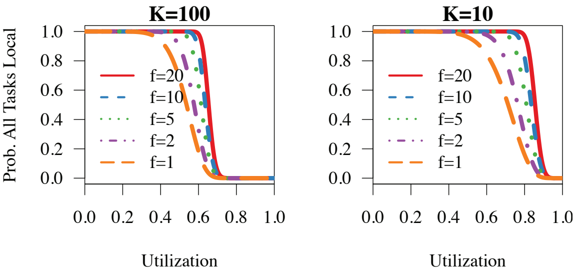


Figure 5: **Probability of input-stage locality when using a sampling function which outputs f disjoint samples. Sampling functions specify additional constraints for samples.**

We mimic job performance with an ideal data-aware scheduler using a "what-if" simulator. Our simulator is unconstrained and (i) assigns memory locality for *all* the tasks in the input phase (we assume 20× speed up for memory locality [45] compared to reading data over the network based on our micro-benchmark) and (ii) places tasks to *perfectly balance* cross-rack links. We see that jobs speed up by 87.6% on average with such ideal data-aware scheduling.

Given these potential benefits, we have designed KMN, a scheduling framework that exploits the available choices to improve performance. At the heart of KMN lie scheduling techniques to increase locality for input (§3) stages and balance network usage for intermediate (§4) stages. In §5, we describe an interface that allows applications to specify all available choices to the scheduler.

3 Input Stage

For the input stage (*i.e.,* the map stage in MapReduce or the extract stage in Dryad) accounting for combinatorial choice leads to improved locality and hence reduced completion time. Here we analyze the improvements in locality in two scenarios: in §3.1 we look at jobs which can use any K of the N input blocks; in §3.2 we look at jobs which use a custom sampling function.

We assume a cluster with s compute slots per machine. Tasks operate on one input block each and input blocks are uniformly distributed across the cluster, this is in line with the block placement policy used by Hadoop. For ease of analysis we assume machines in the cluster are uniformly utilized (*i.e.,* there are no hot-spots). In our evaluation §6) we consider hot-spots due to skewed input-block and machine popularity.

3.1 Choosing any K out of N blocks

Many modern systems *e.g.,* BlinkDB [5], Presto [29], AQUA [2] operate by choosing a random subset of blocks from shuffled input data. These systems rely on the observation that block sampling [20] is statistically equivalent to uniform random sampling (page 243 in [55]) when each block is itself a random sample of the overall population. Given a sample size K, these systems can operate on any K input blocks *i.e.,* for an input of size N the scheduler can choose any one of $\binom{N}{K}$ combinations.

In the cluster setup described above, the probability that a task operating on an input block gets locality is $p_t = 1 - u^s$ where u is the cluster utilization (probability that all slots in a machine are busy is $= u^s$). For such a cluster the probability for K out of N tasks getting locality is given by the binomial CDF function with the probability of success $= p_t$, i.e., $1 - \sum_{i=0}^{K-1} \binom{N}{i} p_t^i (1-p_t)^{N-i}$.

The dominant factor in this probability is the ratio between K and N. In Figure 4 we fix the number of slots per machine to $s = 8$ and plot the probability of $K = 10$ and $K = 100$ tasks getting locality in a job with varying input size N and varying cluster utilization. We observe that the probability of achieving locality is high even when 90% of the cluster is utilized. We also compare this to a baseline that does not exploit this combinatorial choice and pre-selects a random K blocks *beforehand*. For the baseline the probability that all tasks are local drops dramatically even with cluster utilization of 60% or less.

3.2 Custom Sampling Functions

Some systems require additional constraints on the samples used and use custom sampling functions. These sampling functions can be used to produce several K-block samples and the scheduler can pick *any* sample. The scheduler is however constrained to use *all* of the K-blocks from one sample. We consider a sampling function that produces f disjoint samples and analyze locality improvements in this setting.

As noted previous, the probability of a task getting locality is $p_t = 1 - u^s$. The probability that all K blocks in a sample get locality is p_t^K. Since the f samples are disjoint (and therefore the probability of achieving locality is independent) the probability that at least one among the f samples can achieve locality is $p_j = 1 -$

$(1 - p_t^K)^f$. Figure 5 shows the probability of $K = 10$ and $K = 100$ tasks achieving locality with varying utilization and number of samples. We see that the probability of achieving locality significantly increases with f. At $f = 5$ we see that small jobs (10 tasks) can achieve complete locality even when the cluster is 80% utilized.

We thus find that accounting for combinatorial choices can greatly improve locality for the input stage. Next we analyze improvements for intermediate stages.

4 Intermediate Stages

Intermediate stages of jobs commonly involve one-to-all (broadcast), many-to-one (coalesce) or many-to-many (shuffle) network transfers [23]. These transfers are network-bound and hence, often slowed down by congested cross-rack network links. As described in §2.2.2, data-aware scheduling can improve performance by better placement of both upstream and downstream tasks to balance the usage of cross-rack network links.

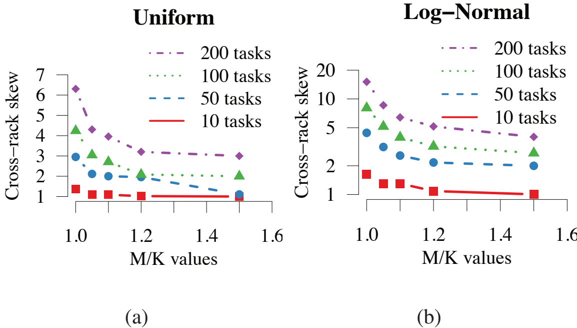
While effective heuristics can be used in scheduling downstream tasks to balance network usage (we deal with this in §5), they are nonetheless limited by the locations of the outputs of upstream tasks. Scheduling upstream tasks to balance the locations of their outputs across racks is often complicated due to many dynamic factors in clusters. First, they are constrained by data locality (§3) and compromising locality is detrimental. Second, the utilization of the cross-rack links when downstream tasks start executing are hard to predict in multi-tenant clusters. Finally, even the size of upstream outputs varies across jobs and are not known beforehand.

We overcome these challenges by scheduling a *few additional* upstream tasks. For an upstream stage with K tasks, we schedule M tasks ($M > K$). Additional tasks increase the likelihood that task outputs are distributed across racks. This allows us to choose the "best" K out of M upstream tasks, out of $\binom{M}{K}$ choices, to minimize cross-rack network utilization. In the rest of this section, we show analytically that a few additional upstream tasks can significantly reduce the imbalance (§4.1). §4.2 describes a heuristic to pick the best K out of M upstream tasks. However, not all M upstream tasks may finish simultaneously because of stragglers; we modify our heuristic to account for stragglers in §4.3.

4.1 Additional Upstream Tasks

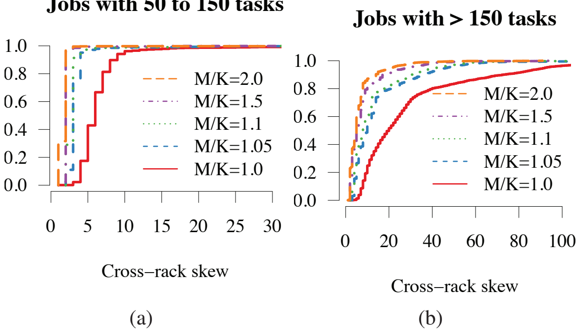
While running additional tasks can balance network usage, it is important to consider how many additional tasks are required. Too many additional tasks can often lead to worsening of overall cluster performance.

We analyze this using a simple model of the scheduling of upstream tasks. For simplicity, we assume that



(a) (b)

Figure 6: **Cross-rack skew as we vary M/K for uniform and log-normal distributions. Even 20% extra upstream tasks greatly reduces network imbalance for later stages.**



(a) (b)

Figure 7: **CDF of cross-rack skew as we vary M/K for the Facebook trace.**

upstream task outputs are equal in size and network links are equally utilized. We only model tasks at the level of racks and evaluate the cross-rack skew (ratio of the rack with largest and smallest number of upstream tasks) using both synthetic distributions of upstream task locations as well as data from our Facebook trace.

Synthetic Distributions: We first consider a scheduler that places tasks on racks uniformly at random. Figure 6(a) plots the cross-rack skew in a 100 rack cluster for varying values of K (*i.e.,* the stage's desired number of tasks) and M/K (*i.e.,* the fraction of additional tasks launched). We can see that even with a scheduler that places the upstream tasks uniformly, there is significant skew for large jobs when there are no additional tasks ($\frac{M}{K} = 1$). This is explained by the balls and bins problem [46] where the maximum imbalance is expected to be $O(logn)$ when distributing n balls.

However, we see that even with 10% to 20% additional tasks ($\frac{M}{K} = 1.1 - 1.2$) the cross-rack skew is reduced by $\geq 2\times$. This is because when the number of upstream tasks, n is > 12, $0.2n > logn$. Thus, we can avoid most of the skew with just a few extra tasks.

We also repeat this study with a log normal distribution ($\theta = 0, m = 1$) of upstream task placement; this is more skewed compared to the uniform distribution.

However, even with a log-normal distribution, we again see that a few extra tasks can be very effective at reducing skew. This is because the expected value of the most loaded bin is still linear and using $0.2n$ additional tasks is sufficient to avoid most of the skew.

Facebook Distributions: We repeat the above analysis using the number and location of upstream tasks of a phase in the Facebook trace (used in §2.2.2). Recall the high cross-rack skew in the Facebook trace. Despite that, again, a few additional tasks suffices to eliminate a large fraction of the skews. Figure 7 plots the results for varying values of $\frac{M}{K}$ for different jobs. A large fraction of the skew is reduced by running just 10% more tasks. This is nearly 66% of the reductions we get using $\frac{M}{K} = 2$.

In summary we see that running a few extra tasks is an effective strategy to reduce skew, both with synthetic as well as real-world distributions. We next look at mechanisms that can help us achieve such reduction.

4.2 Selecting Best Upstream Outputs

The problem of selecting the best K outputs from the M upstream tasks can be stated as follows: We are given M upstream tasks $U = u_1 ... u_M$, R downstream tasks $D = d_1 ... d_R$ and their corresponding rack locations. Let us assume that tasks are distributed over racks $1...L$ and let $U' \subset U$ be some set of K upstream outputs. Then for each rack we can define the uplink cost C_{2i-1} and downlink cost C_{2i} using a cost function $C_i(U', D)$. Our objective then is to select U' to minimize the most loaded link i.e.

$$\arg\min_{U'} \max_{i \in 2L} C_i(U', D)$$

While this problem is NP-Hard [57], many approximation heuristics have been developed. We use a heuristic that corresponds to spreading our choice of K outputs across as many racks as possible.[1]

Our implementation for this approximation heuristic is shown in Algorithm 1. We start with the list of upstream tasks and build a hash map that stores how many tasks were run on each rack. Next we sort the tasks first by their index within a rack and then by the number of tasks in the rack. This sorting criteria ensures that we first see one task from each rack, thus ensuring we spread our choices across racks. We use an additional heuristic of favoring racks with more outputs to help our downstream task placement techniques (§5.2.2). The main computation cost in this method is the sorting step and hence this runs in $O(MlogM)$ time for M tasks.

[1]This problem is an instance of the facility location problem [26] where we have a set of clients (downstream tasks), set of potential facility locations (upstream tasks), a cost function that maps facility locations to clients (link usage). Our heuristic follows from picking a facility that is farthest from the existing set of facilities [30].

Algorithm 1 Choosing K upstream outputs out of M using a round-robin strategy

```
1: Given: upstreamTasks - list with rack, index within rack
   for each task
2: Given: K - number of tasks to pick
3: // Number of upstream tasks in each rack
4: upstreamRacksCount = map()
5:
6: // Initialize
7: for task in upstreamTasks do
8:     upstreamRacksCount[task.rack] += 1
9: end for
10:
11: // Sort the tasks in round-robin fashion
12: roundRobin = upstreamTasks.sort(CompareTasks)
13: chosenK = roundRobin[0 : K]
14: return chosenK
15:
16: procedure CompareTasks(task1,task2)
17:     if task1.idx != task2.idx then
18:         // Sort first by index
19:         return task1.idx < task2.idx
20:     else
21:         // Then by number of outputs
22:         numRack1 = upstreamRacksCount[task1.rack]
23:         numRack2 = upstreamRacksCount[task2.rack]
24:         return numRack1 > numRack2
25:     end if
26: end procedure
```

4.3 Handling Upstream Stragglers

While the previous section described a heuristic to pick the best K out of M upstream outputs, waiting for all M can be inefficient due to *stragglers*. Stragglers in the upstream stage can delay completion of some tasks which cuts into the gains obtained by balancing the network links. Stragglers are a common occurrence in clusters with many clusters reporting significantly slow tasks despite many prevention and speculation solutions [10, 13, 61]. This presents a trade-off in waiting for all M tasks and obtaining the benefits of choice in picking upstream outputs against the wasted time for completion of all M upstream tasks including stragglers. Our solution for this problem is to schedule downstream tasks at some point after K upstream tasks have completed but not wait for the stragglers in the M tasks. We quantify this trade-off with analysis and micro-benchmarks.

4.3.1 Stragglers vs. Choice

We study the impact of stragglers in the Facebook trace when we run 2%, 5% and 10% extra tasks (i.e., $\frac{M}{K} = 1.02, 1.05, 1.1$). We compute the difference between the time taken for the fastest K tasks and the time to com-

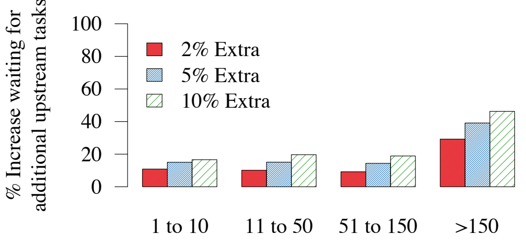


Figure 8: **Percentage of time spent waiting for additional upstream tasks when running 2%, 5% or 10% extra tasks. Stage completion time can be increased by up** $20\% - 40\%$ **due to stragglers.**

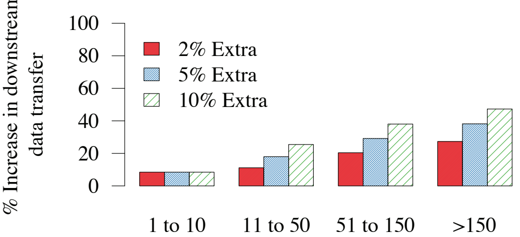


Figure 9: **Percentage of additional time spent in downstream data transfer when not using choices from 2%, 5% or 10% extra tasks. Decrease in choice increases data transfer time by** $20\% - 40\%$**.**

plete all M tasks. Figure 8 shows that waiting for the extra tasks can inflate the completion of the upstream phase by $20\% - 40\%$ (for jobs with > 150 tasks). Also, the trend of using a large number of small tasks [48] for interactive jobs will only worsen such inflation. On the other hand avoiding upstream stragglers by using the fastest tasks reduces the available choice. Consequently, the time taken for downstream data transfer increases. The lack of choices from extra tasks means we cannot balance network usage. Figure 9 shows that not using choice from additional tasks can increase data transfer time by 20% for small jobs (11 to 50 tasks) and up to 40% for large jobs (> 150 tasks). We now devise a simple approach to balance between the above two factors—waiting for upstream stragglers versus losing choice for downstream data transfer.

4.3.2 Delayed Stage Launch

The problem we need to solve can be formulated as: we have M upstream tasks $u_1, u_2, ..., u_M$ and for each task we have corresponding rack locations. Our goal is to find the optimal *delay* after the first K tasks have finished, such that the overall time taken is minimized. In other words, our goal is to find the optimal K' tasks to wait for before starting the downstream tasks.

We begin with assuming an oracle that can give us the task finish times for all the tasks. Given such a oracle we can sort the tasks in an increasing order of finish times such that $F_j \geq F_i \; \forall j > i$. Let us define the waiting delay for tasks $K+1$ to M as $D_i = F_i - F_k \; \forall i > k$. We also assume that given K' tasks, we can compute the optimal K tasks to use (§4.2) and the estimated transfer time $S_{K'}$.

Our problem is to pick K' ($K \leq K' \leq M$) such that the total time for the data transfer is minimized. That is we need to pick K' such that $F_k + D_{k'} + S_{k'}$ is minimized. In this equation F_k is known and independent of K'. Of the other two, $D_{k'}$ increases as k' goes from K to M, while $S_{k'}$ decreases. However as the sum of an increasing and decreasing function is not necessarily convex [2] it isn't easy to minimize the total time taken.

Delay Heuristic: While the brute-force approach would require us to try all values from K to M, we develop two heuristics that allow us to bound the search space and quickly find the optimal value of K'.

- *Bounding transfer*: At the beginning of the search procedure we find the maximum possible improvement we can get from picking the best set of tasks. Whenever the delay $D_{K'}$ is greater than the maximum improvement, we can stop the search as the succeeding delays will increase the total time.
- *Coalescing tasks*: We can also coalesce a number of task finish events to further reduce the search space. For example we can coalesce task finish events which occur close together by time i.e., cases $D_{i+1} - Di < \delta$. This will mean our result is off by at most δ from the optimal, but for small values of δ we can coalesce tasks of a wave that finish close to each other.

Using these heuristics we can find the optimal number of tasks to wait for quickly. For example, in the Facebook trace described before using $M/K = 1.1$ or 10% extra tasks, determining the optimal wait time for a job requires looking at less than 4% of all configurations when we use a coalescing error of 1%. We found coalescing tasks to be particularly useful as even with a δ of 0.1% we need to look at around 8% of all possible configurations. Running without any coalescing is infeasible since it takes ≈ 1000 ms.

Finally, we relax our assumption of an oracle as follows. While the task finish times are not exactly known beforehand, we use job sizes to figure out if the same job has been run before. Based on this we use the job history to predict the task finish times. This approach should work well for clusters that have many jobs run periodically [36]. In case the job history is not available we can fit the tasks length distribution using the first few task finish times and use that to get approximate task finish times for the rest of the tasks [28].

[2] Take any non-convex function and make its increasing region F_i and its decreasing region F_d and it can be seen that the sum isn't convex.

```
// SQL Query
SELECT status, SUM(quantity)
FROM items
GROUP BY status

// Spark Query
kv = file.map{ li =>
  (li.l_linestatus,li.quantity)}
result = kv.reduceByKey{(a,b) =>
  a + b}.collect()

// KMN Query
sample = file.blockSample(0.1, sampler=None)
kv = sample.map{ li =>
  (li.l_linestatus,li.quantity)}
result = kv.reduceByKey{(a,b) =>
  a + b}.collect()
```

Figure 10: **An example of a query in SQL, Spark and** KMN

5 System Implementation

We have built KMN on top of Spark [60], an open-source cluster computing framework. Our implementation is based on Spark version 0.7.3 and KMN consists of 1400 lines of Scala code. In this section we discuss the features of our implementation and implementation challenges.

5.1 Application Interface

We define a `blockSample` operator which jobs can use to specify input constraints (for instance, use K blocks from file F) to the framework.The `blockSample` operator takes two arguments: the ratio $\frac{K}{N}$ and a *sampling function* that can be used to impose constraints. The sampling function can be used to choose user-defined sampling algorithms (*e.g.,* stratified sampling). By default the sampling function picks any K of N blocks.

Consider an example SQL query and its corresponding Spark [60] version shown in Figure 10. To run the same query in KMN we just need to prefix the query with the `blockSample` operator. The *sampler* argument is a Scala closure and passing `None` causes the scheduler to use the default function which picks any K out of the N input blocks. This design can be readily adapted to other systems like Hadoop MapReduce and Dryad.

KMN also provides an interface for jobs to introspect which samples where used in a computation. This can be used for error estimation using algorithms like Bootstrap [4] and also provides support for queries to be repeated. We implement this in KMN by storing the K partitions used during computation as a part of a job's lineage. Using the lineage also ensures that the same samples are used if the job is re-executed during fault recovery [60].

5.2 Task Scheduling

We modify Spark's scheduler in KMN to implement the techniques described in earlier sections.

5.2.1 Input Stage

Schedulers for frameworks like MapReduce or Spark typically use a slot-based model where the scheduler is invoked whenever a slot becomes available in the cluster. In KMN, to choose any K out of N blocks we modify the scheduler to run tasks on blocks local to the first K available slots. To ensure that tasks don't suffer from resource starvation while waiting for locality, we use a timeout after which tasks are scheduled on any available slot. Note that, choosing the first K slots provides a sample similar or slightly better in quality compared to existing systems like Aqua [2] or BlinkDB [5] that reuse samples for short time periods. To schedule jobs with custom sampling functions, we similarly modify the scheduler to choose among the available samples and run the computation on the sample that has the highest locality.

5.2.2 Intermediate Stage

Existing cluster computing frameworks like Spark and Hadoop place intermediate stages without accounting for their dependencies. However smarter placement which accounts for a tasks' dependencies can improve performance. We implemented two strategies in KMN:

Greedy assignment: The number of cross-rack transfers in the intermediate stage can be reduced by co-locating map and reduce tasks (more generally any dependent tasks). In the greedy placement strategy we maximize the number of reduce tasks placed in the rack with the most map tasks. This strategy works well for small jobs where network usage can be minimized by placing all the reduce tasks in the same rack.

Round-robin assignment: While greedy placement minimizes the number of transfers from map tasks to reduce tasks it results in most of the data being sent to one or a few racks. Thus the links into these racks are likely to be congested. This problem can be solved by distributing tasks across racks while simultaneously minimizing the amount of data sent across racks. This can be achieved by evenly distributing the reducers across racks with map tasks. This strategy can be shown to be optimal if we know the map task locations and is similar in nature to the algorithm described in §4.2. We perform a more detailed comparison of the two approaches in §6

5.3 Support for extra tasks

One consequence of launching extra tasks to improve performance is that the cluster utilization could be af-

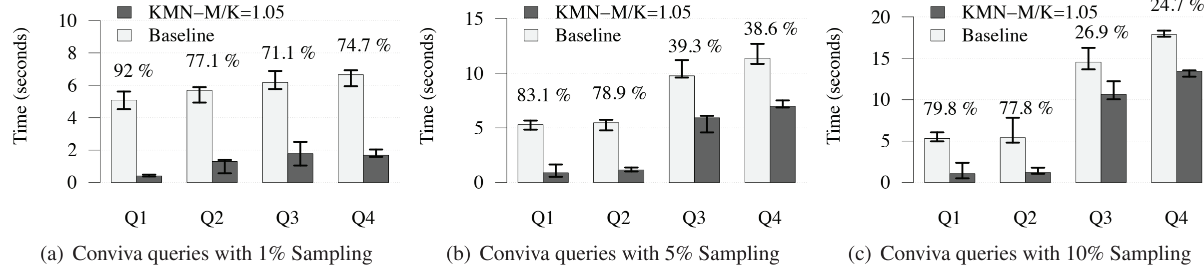


Figure 11: **Comparing baseline and KMN-1.05 with sampling-queries from Conviva. Numbers on the bars represent percentage improvement when using KMN-$M/K = 1.05$.**

fected by these extra tasks. To avoid utilization spikes, in KMN the value for M/K (the percentage of extra tasks to launch) can only be set by the cluster administrator and not directly by the application. Further, we implemented support for killing tasks once the scheduler decides that the tasks' output is not required. Killing tasks in Spark is challenging as tasks are run in threads and many tasks share the same process. To avoid expensive clean up associated with killing threads [1], we modified tasks in Spark to periodically poll and check a status bit. This means that tasks sometimes could take a few seconds more before they are terminated, but we found that this overhead was negligible in practice.

In KMN, using extra tasks is crucial in extending the flexibility of many choices throughout the DAG. In §3 and §4 we discussed how to use the available choices in the input and intermediate stages in a DAG. However, jobs created using frameworks like Spark or DryadLINQ can extend across many more stages. For example, complex SQL queries may use a map followed a shuffle to do a group-by operation and follow that up with a join. One solution to this would be run more tasks than required in every stage to retain the ability to choose among inputs in succeeding stages. However we found that in practice this does not help very much. In frameworks like Spark which use lazy evaluation, every stage following than the first stage is treated as an intermediate stage. As we use a round-robin strategy to schedule intermediate tasks (§5.2.2), the outputs from the first intermediate stage are already well spread out across the racks. Thus there isn't much skew across racks that affects the performance of following stages. In evaluation runs we saw no benefits for later stages of long DAGs.

6 Evaluation

We evaluate the benefits of KMN using two approaches: first we run approximate queries used in production at Conviva, a video analytics company, and study how KMN compares to using existing schedulers with pre-selected samples. Next we analyze how KMN behaves in a shared cluster, by replaying a workload trace obtained from Facebook's production Hadoop cluster.

Metric: In our evaluation we measure percentage improvement of job completion time when using KMN. We define percentage improvement as:

$$\% \text{ Improvement} = \frac{\text{Baseline Time} - \text{KMN Time}}{\text{Baseline Time}} \times 100$$

Our evaluation shows that,

- KMN improves real-world sampling-based queries from Conviva by more than 50% on average across various sample sizes and machine learning workloads by up to 43%.
- When replaying the Facebook trace, on an EC2 cluster, KMN can improve job completion time by 81% on average (92% for small jobs)
- By using 5% – 10% extra tasks we can balance bottleneck link usage and decrease shuffle times by 61% – 65% even for jobs with high cross-rack skew.

6.1 Setup

Cluster Setup:We run all our experiments using 100 m2.4xlarge machines on Amazon's EC2 cluster, with each machine having 8 cores, 68GB of memory and 2 local drives. We configure Spark to use 4 slots and 60 GB per machine. To study memory locality we cache the input dataset before starting each experiment. We compare KMN with a baseline that operates on a pre-selected sample of size K and does not employ any of the shuffle improvement techniques described in §4, §5. We also label the fraction of extra tasks run (*i.e.,* M/K), so KMN-$M/K = 1.0$ has $K = M$ and KMN-$M/K = 1.05$ has 5% extra tasks. Finally, all experiments were run at least three times and we plot median values across runs and use error bars to show minimum and maximum values.

Workload: Our evaluation uses a workload trace from Facebook's Hadoop cluster [21]. The traces are from a mix of interactive and batch jobs and capture over half a million jobs on a 3500 node cluster. We use a scaled down version of the trace to fit within our cluster and use the same inter-arrival times and the task-to-rack mapping

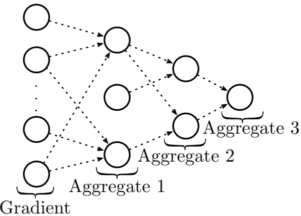


Figure 12: **Execution DAG for Stochastic Gradient Descent (SGD).**

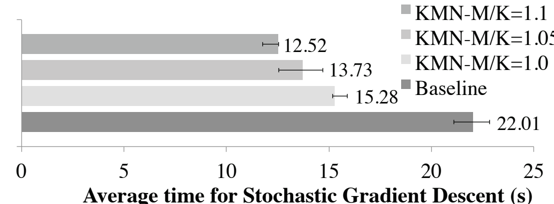


Figure 13: **Overall improvement when running Stochastic Gradient Descent using KMN**

as in the trace. Unless specified, we use 10% sampling when running KMN for all jobs in the trace.

We begin by showing overall gains with KMN (§6.2), then present benefits for input stages from KMN (§6.3) and finally show how KMN affects intermediate stages (§6.4).

6.2 Benefits of KMN

We evaluate the benefits of using KMN on three workloads: real-world approximate queries from Conviva, a machine learning workload running Stochastic Gradient Descent and a Hadoop workload trace from Facebook.

6.2.1 Conviva Sampling jobs

We first present results from running 4 real-world sampling queries obtained from Conviva, a video analytics company. The queries were run on access logs obtained across a 5-day interval. We treat the entire data set as N blocks and vary the sampling fraction (K/N) to be 1%, 5% and 10%. We run the queries at 50% cluster utilization and run each query multiple times.

Figure 11 shows the median time taken for each query and we compare KMN-$M/K = 1.05$ to the baseline that uses pre-selected samples. For query 1 and query 2 we can see that KMN gives 77%–91% win across 1%, 5% and 10% samples. Both these queries calculate summary statistics across a time window and most of the computation is performed in the map stage. For these queries KMN ensures that we get memory locality and this results in significant improvements. For queries 3 and 4, we see around 70% improvement for 1% samples, and this reduces to around 25% for 10% sampling. Both

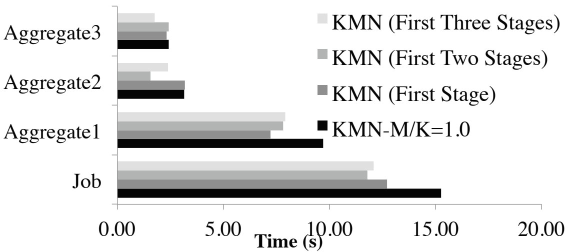


Figure 14: **Breakdown of aggregation times when using KMN for different number of stages in SGD**

Job Size	% Overall	% Map Stage	% Shuffle
1 to 10	92.8	95.5	84.61
11 to 100	78	94.1	28.63
> 100	60.8	95.4	31.02

Table 1: **Improvements over baseline, by job size and stage**

these queries compute the number of distinct users that match a specified criteria. While input locality also improves these queries, for larger samples the reduce tasks are CPU bound (while they aggregate values).

6.2.2 Machine learning workload

Next, we look at performance benefits for a machine learning workload that uses sampling. For our analysis, we use Stochastic Gradient Descent (SGD). SGD is an iterative method that scales to large datasets and is widely used in applications such as machine translation and image classification. We run SGD on a dataset containing 2 million data items, where each each item contains 4000 features. The complete dataset is around 64GB in size and each of our iterations operates on a 1% sample 1% of the data. Thus the random sampling step reduces the cost of gradient computation by 100× but maintains rapid learning rates [52]. We run 10 iterations in each setting to measure the total time taken for SGD.

Each iteration consists of a DAG comprised of a map stage where the gradient is computed on sampled data items and the gradient is then aggregated from all points. The aggregation step can be efficiently performed by using an aggregation tree as shown in Figure 12. We implement the aggregation tree using a set of shuffle stages and use KMN to run extra tasks at each of these aggregation stages.

The overall benefits from using KMN are shown in Figure 13. We see that KMN-$M/K = 1.1$ improves performance by 43% as compared to the baseline. These improvements come from a combination of improving memory locality for the first stage and by improving shuffle performance for the aggregation stages. We further break down the improvements by studying the effects of KMN at every stage in Figure 14.

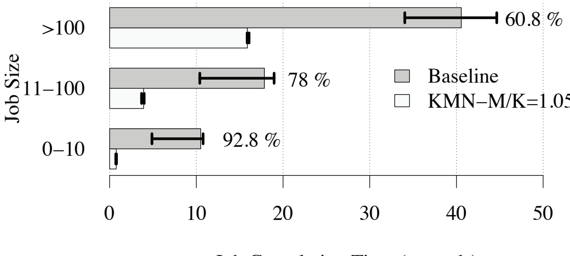


Figure 15: **Overall improvement from KMN compared to baseline. Numbers on the bar represent percentage improvement using KMN-$M/K = 1.05$.**

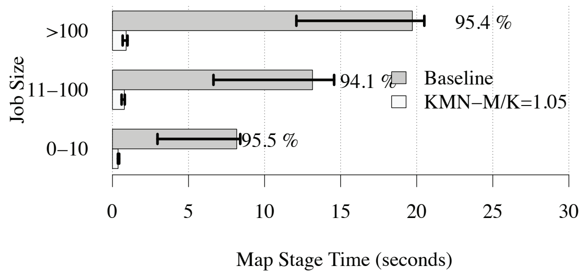


Figure 16: **Improvement due to memory locality for the Map Stage for the Facebook trace. Numbers on the bar represent percentage improvement using KMN-$M/K = 1.05$.**

When running extra tasks for only the first stage (gradient stage), we see improvements of around 26% for the first aggregation (Aggregate 1); see KMN (First Stage). Without extra tasks the next two aggregation stages (Aggregate 2 and Aggregate 3) behave similar to KMN-$M/K = 1.0$. When extra tasks are spawned for later stages too, benefits propagate and we see 50% improvement in the second aggregation (Aggregate-2) while using KMN for the first two stages. However, propagating choice across stages does impose some overheads. Thus even though we see that KMN (First Three Stages) improves the performance of the last aggregation stage (Aggregate 3), running extra tasks slows down the overall job completion time (Job). This is because the final aggregation steps usually have fewer tasks with smaller amounts of data, which makes running extra tasks not worth the overhead. We plan to investigate techniques to estimate this trade-off and automatically determine which stages to use KMN for in the future.

6.2.3 Facebook workload

We next quantify the overall improvements across the trace from using KMN. To do this, we use a baseline configuration that mimics task locality from the original trace while using pre-selected samples. We compare this to KMN-$M/K = 1.05$ that uses 5% extra tasks and a round-robin reducer placement strategy (§5.2.2). The results showing average job completion time broken down

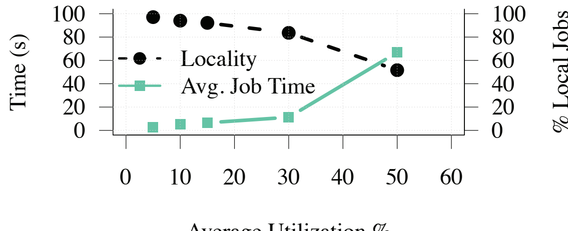


Figure 17: **Job completion time and locality as we increase utilization.**

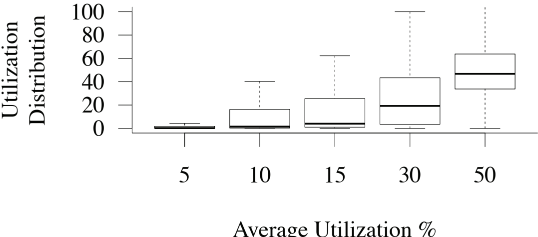


Figure 18: **Boxplot showing utilization distribution for different values of average utilization.**

by job size is shown in Figure 15 and relative improvements are shown in Table 1. As seen in the figure, using KMN leads to around 92% improvement for small jobs with < 10 tasks and more than 60% improvement for all other jobs. Across all jobs KMN-$M/K = 1.05$ improves performance by 81%, which is 93% of the potential win (§2.3).

To quantify where we get improvements from, we break down the time taken by different stages of a job. Improvements for the input stage or the map stage are shown in Figure 16. We can see that using KMN we are able to get memory locality for almost all the jobs and this results in around 94% improvement in the time taken for the map stage. This is consistent with the predictions from our model in §3 and shows that pushing down sampling to the run-time can give tremendous benefits. The improvements in the shuffle stage are shown in Figure 19. For small jobs with < 10 tasks we get around 85% improvement and these are primarily because we co-locate the mappers and reducers for small jobs and thus avoid network transfer overheads. For large jobs with > 100 tasks we see around 30% improvement due to reduction in cross-rack skew.

6.3 Input Stage Locality

Next, we attempt to measure how the locality obtained by KMN changes with cluster utilization. As we vary the cluster utilization, we measure the average job completion time and fraction of jobs where all tasks get locality. The results shown in Figure 17 show that for up to 30% average utilization, KMN ensures that more than 80% of

Job Size	$M/K = 1.0$	$M/K = 1.05$	$M/K = 1.1$
1 to 10	85.04	84.61	83.76
11 to 100	27.5	28.63	28.18
> 100	14.44	31.02	36.35

Table 2: **Shuffle time improvements over baseline while varying M/K**

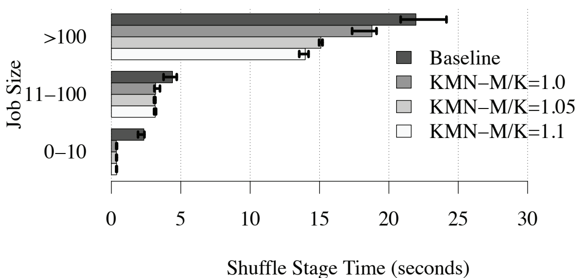


Figure 19: **Shuffle improvements when running extra tasks.**

Cross-rack skew	M/K=1.0	$M/K = 1.05$	$M/K = 1.1$
≤ 4	24.45	29.22	30.81
4 to 8	15.26	27.60	33.92
≥ 8	48.31	61.82	65.82

Table 3: **Shuffle improvements with respect to baseline as cross-rack skew increases.**

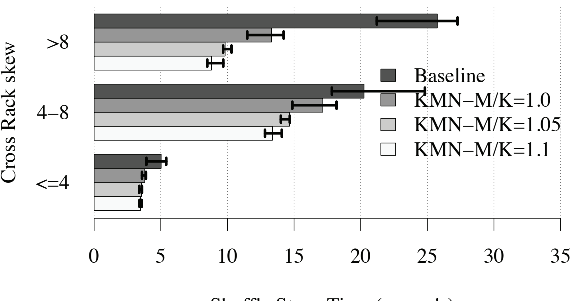


Figure 20: **Difference in shuffle performance as cross-rack skew increases**

jobs get perfect locality. We also observed significant variance in the utilization during the trace replay and the distribution of utilization values is shown as a boxplot in Figure 18. From this figure we can see that while average utilization is 30% we observe utilization spikes of up to 90%. Because of such utilization spikes, we see periods of time where all jobs do not get locality.

Finally, at 50% average utilization (utilization spikes $> 90\%$) only around 45% of jobs get locality. This is lower than predictions from our model in §3. There are two reasons for this difference: First, our experimental cluster has only 400 slots and as we do 10% sampling ($K/N = 0.1$), the setup doesn't have enough choices for jobs with > 40 map tasks. Further the utilization spikes also are not taken into account by the model and jobs which arrive during a spike do not get locality.

6.4 Intermediate Stage Scheduling

In this section we evaluate scheduling decisions by KMN for intermediate stages. First we look at the benefits from running additional map tasks and then evaluate the delay heuristic used for straggler mitigation. Finally we also measure KMN's sensitivity to reducer placement strategies.

6.4.1 Effect of varying M/K

We evaluate the effect of running extra map tasks (i.e $M/K > 1.0$) and measure how that influences the time taken for shuffle operations. For this experiment we wait until all the map tasks have finished and then calculate the best reducer placement and choose the best K map outputs as per techniques described in §4.2. The average time for the shuffle stage for different job sizes is shown in Figure 19 and the improvements with respect to the baseline are shown in Table 2. From the figure, we see that for small jobs with less than 10 tasks there is almost no improvement from running extra tasks as they usually do not suffer from cross-rack skew. However for large jobs with more than 100 tasks, we now get up to 36% improvement in shuffle time over the baseline.

Further, we can also analyze how the benefits are sensitive to the cross-rack skew. We plot the average shuffle time split by cross-rack skew in Figure 20. Correspondingly we list the improvements over the baseline in Table 3. We can see that for jobs which have low cross-rack skew, we get up to 33% improvement when using KMN-$M/K = 1.1$. Further, for jobs which have cross-rack skew > 8, we get up to 65% improvement in shuffle times and a 17% improvement over $M/K = 1$.

6.4.2 Delayed stage launch

We next study the impact of stragglers and the effect of using the delayed stage launch heuristic from §4.3. We run the Facebook workload at 30% cluster utilization with KMN-$M/K = 1.1$ and compare our heuristic to two baseline strategies. In one case we wait for the first K map tasks to finish before starting the shuffle while in the other case we wait for all M tasks for finish. The performance break down for each stage is shown in Figure 21. From the figure we see that for small jobs (< 10 tasks) which don't suffer from cross-rack skew, KMN performs similar to picking the first K map outputs. This is because in this case stragglers dominate the shuffle wins possible from using extra tasks. For larger tasks we see that our heuristic can dynamically adjust the stage delay to ensure we avoid stragglers while getting the benefits of balanced

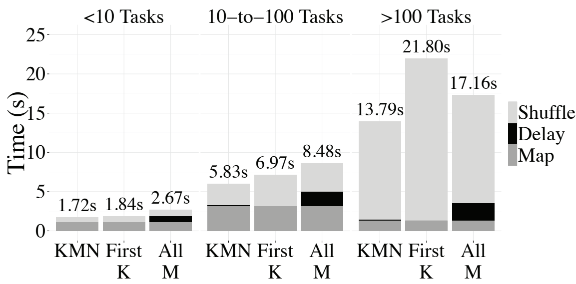


Figure 21: **Benefits from straggler mitigation and delayed stage launch.**

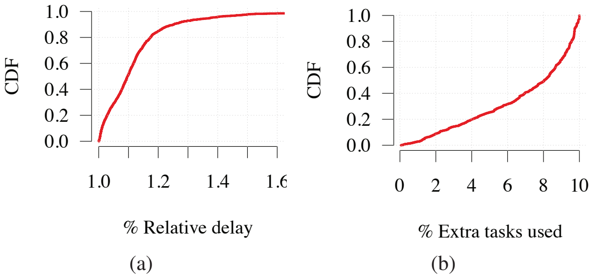


Figure 22: **(a) CDF of % time that the job was delayed (b) CDF of % of extra map tasks used.**

shuffle operations. For example for jobs with > 10 tasks KMN adds $5\% - 14\%$ delay after first K tasks complete and still gets most of the shuffle benefits. Overall, this results in an improvement of up to 35%.

For more fine-grained analysis we also ran an event-driven simulation that uses task completion times from the same Facebook trace. The CDF of extra map tasks used is shown in Figure 22(b), where we see that around 80% of the jobs wait for 5% or more map tasks. We also measured the time relative to when the first K map tasks finished and to normalize the delay across jobs we compute the relative wait time. Figure 22(a) shows the CDF of relative wait times and we see that the delay is less than 25% for 62% of the jobs. The simulation results again show that our relative delay is not very long and that job completion time can be improved when we use extra tasks available within a short delay.

6.4.3 Sensitivity to reducer placement

To evaluate the importance of reduce placement strategy, we compare the time taken for the shuffle stage for the round-robin strategy described in §5.2.2 against a greedy assignment strategy that attempts to pack reducers into as few machines as possible. Note that the baseline used in our earlier experiments used a random reducer assignment policy and §6.2.3 compares the round-robin strategy to random assignment. Figure 23 shows the results

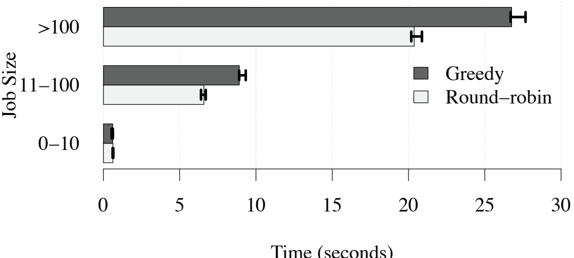


Figure 23: **Difference between using greedy assignment of reducers versus using a round-robin scheme to place reducers among racks with upstream tasks.**

from this experiment with the results broken down by job size. From the results we can see that for jobs with > 10 tasks using a round-robin placement can improve performance by 10%-30%. However for very small jobs, running tasks on more machines increases the variance and the greedy assignment in fact performs 8% better.

7 Related Work

Cluster schedulers: Cluster scheduling has been an area of active research and recent work has proposed techniques to enforce fairness [32, 39], satisfy job constraints [33] and improve locality [39, 59]. In KMN, we focus on applications that have input choices and propose techniques to exploit the available flexibility while scheduling tasks. Straggler mitigation solutions launch extra copies of tasks to mitigate the impact of slow running tasks [10, 12, 61]. While KMN shares the similarity of executing extra copies, our goals are different. Further, straggler mitigation solutions are limited by the number of replicas of the input data, and can leverage our observation of combinatorial choices towards more effective speculation. Prior efforts in improving shuffle performance [7, 24] have looked at either provisioning the network better or scheduling flows to improve performance. On the other hand, in KMN we use additional tasks and better placement techniques to balance data transfers across racks. Finally, recent work [49] has also looked at using the power of many choices to reduce scheduling latency. In KMN we exploit the power choices to improve network balance using just a few additional tasks.

Approximate Query Processing Systems: Approximate query processing (AQP) systems such as Aqua [2], STREAM [47], and BlinkDB [5] use pre-computed samples to answer queries. These works are complimentary to our work, and we expect that projects like BlinkDB can use KMN to improve performance, while maintaining, or in some cases even improving response quality. Prior work in databases has also proposed Online Aggregation [37] (OLA) methods that can be used to

present approximate aggregation results while the input data is processed in a streaming fashion. Recent extensions [25, 51] have also looked at supporting OLA-style computations in MapReduce. In contrast, KMN can be used for scheduling sampling applications which do not process the entire dataset and process a fixed and small sample of data.

Machine learning frameworks: Recently, a large body of work has focused on building cluster computing frameworks that support machine learning tasks. Examples include GraphLab [34, 44], Spark [60], DistBelief [27], and MLBase [41]. Of these, GraphLab and Spark add support for abstractions commonly used in machine learning. Neither of these frameworks provide any explicit system support for sampling. For instance, while Spark provides a sampling operator, this operation is carried out entirely in application logic, and the Spark scheduler is oblivious to the use of sampling.

8 Conclusion

The rapid growth of data stored in clusters, increasing demand for interactive analysis, and machine learning workloads have made it inevitable that applications will operate on subsets of data. It is therefore imperative that schedulers for cluster computing frameworks exploit the available choices to improve performance. As a first step towards this goal we have presented KMN, a system that improves data-aware scheduling for jobs with combinatorial choices. Using our prototype implementation, we have shown that KMN can improve performance by increasing locality and balancing intermediate data transfers.

Acknowledgments

We are indebted to Ali Ghodsi, Kay Ousterhout, Colin Scott, Peter Bailis, the various reviewers and our shepherd Yuanyuan Zhou for their insightful comments and suggestions. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adobe, Apple, Inc., Bosch, C3Energy, Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, and Yahoo!.

References

[1] Why is Thread.stop deprecated. `http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html`.

[2] S. Acharya, P. Gibbons, and V. Poosala. Aqua: A fast decision support systems using approximate query answers. In *Proceedings of the International Conference on Very Large Data Bases*, pages 754–757, 1999.

[3] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica. Blink and it's done: interactive queries on very large data. *Proceedings of the VLDB Endowment*, 5(12):1902–1905, 2012.

[4] S. Agarwal, H. Milner, A. Kleiner, A. Talwarkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing When Youre Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*. ACM, 2014.

[5] S. Agarwal, B. Mozafari, A. Panda, M. H., S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th European conference on Computer Systems*. ACM, 2013.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM 2008*, Seattle, WA.

[7] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[8] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the European conference on Computer systems (Eurosys'11)*, pages 287–300. ACM, 2011.

[9] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 12–12. USENIX Association, 2011.

[10] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *USENIX NSDI*, 2012.

[12] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. Grass: Trimming stragglers in approximation analytics. NSDI, 2014.

[13] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010.

[14] Apache Hadoop NextGen MapReduce (YARN). Retrieved 9/24/2013, URL: `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[15] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proceedings of ACM SIGCOMM 2012*, pages 431–442. ACM, 2012.

[16] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, August 2010. Springer.

[17] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, volume 20, pages 161–168. NIPS Foundation, 2008.

[18] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, Prague, Czech Republic, June 2007.

[19] M. Cafarella, E. Chang, A. Fikes, A. Halevy, W. Hsieh, A. Lerner, J. Madhavan, and S. Muthukrishnan. Data management projects at Google. *SIGMOD Record*, 37(1):34–38, Mar. 2008.

[20] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 287–298. ACM, 2004.

[21] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[22] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of ACM SIGCOMM 2013*, pages 231–242, Hong Kong, China, 2013.

[23] M. Chowdhury and I. Stoica. Coflow: a networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[24] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 98–109, Toronto, Ontario, Canada, 2011. ACM.

[25] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[26] G. Cornuejols, G. L. Nemhauser, and L. A. Wolsey. The uncapacitated facility location problem. Technical report, DTIC Document, 1983.

[27] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1232–1240, 2012.

[28] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. ASPLOS, 2014.

[29] Facebook Presto. Retrieved 9/21/2013, URL: `http://gigaom.com/2013/06/06/facebook-unveils-presto-engine-for-querying-250-pb-data-warehouse/`.

[30] T. Feder and D. Greene. Optimal algorithms for approximate clustering. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 434–444, Chicago, Illinois, USA, 1988. ACM.

[31] J. Gantz and D. Reinsel. Digital universe study: Extracting value from chaos, 2011.

[32] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX NSDI*, 2011.

[33] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th European conference on Computer Systems*. ACM, 2013.

[34] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. of the 10th USENIX conference on Operating systems design and implementation*, 2012.

[35] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM 2009*, Barcelona, Spain.

[36] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou. Wave computing in the cloud. In *HotOS*, 2009.

[37] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *ACM SIGMOD Record*, volume 26, pages 171–182. ACM, 1997.

[38] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *Proc. of the 2nd European Conference on Computer Systems*, 41(3), 2007.

[39] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[40] P. Kolari, A. Java, T. Finin, T. Oates, and A. Joshi. Detecting spam blogs: A machine learning approach. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1351, 2006.

[41] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. J. Franklin, and M. Jordan. MLbase: A distributed machine-learning system. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[42] J. Langford. The Ideal Large-Scale Machine Learning Class. `http://hunch.net/?p=1729`.

[43] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1028–1039, 2012.

[44] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 2012.

[45] J. McCalpin. STREAM update for Intel Xeon Phi SE10P. `http://www.cs.virginia.edu/stream/stream_mail/2013/0015.html`.

[46] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[47] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. CIDR, 2003.

[48] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. HotOS, 2013.

[49] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[50] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 2013.

[51] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for Large Mapreduce Jobs. *PVLDB*, 4(11):1135–1145, 2011.

[52] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[53] N. Schraudolph, J. Yu, and S. Günter. A stochastic quasi-newton method for online convex optimization. *Journal of Machine Learning Research*, 2:428–435, 2007.

[54] L. Sidirourgos, M. Kersten, and P. Boncz. Sciborq: Scientific Data Management with Bounds on Runtime and Quality. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, pages 296–301, 2011.

[55] P. Sukhatme and B. Sukhatme. *Sampling theory of surveys: with applications*. Asia Publishing House, 1970.

[56] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan. Scale-Out Networking in the Data Center. *IEEE Micro*, 30(4):29–41, July 2010.

[57] J. Vygen. Approximation algorithms facility location problems. Technical Report 05950, Research Institute for Discrete Mathematics, University of Bonn, 2005.

[58] R. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, 2013.

[59] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.

[60] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[61] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, pages 29–42, 2008.

Heading Off Correlated Failures through Independence-as-a-Service

Ennan Zhai†, Ruichuan Chen§, David Isaac Wolinsky†, Bryan Ford†
†*Yale University* §*Bell Labs / Alcatel-Lucent*

Abstract

Today's systems pervasively rely on redundancy to ensure reliability. In complex multi-layered hardware/software stacks, however – especially in the clouds where many independent businesses deploy interacting services on common infrastructure – seemingly independent systems may share deep, hidden dependencies, undermining redundancy efforts and introducing unanticipated correlated failures. Complementing existing post-failure forensics, we propose *Independence-as-a-Service* (or INDaaS), an architecture to audit the independence of redundant systems proactively, thus avoiding correlated failures. INDaaS first utilizes pluggable dependency acquisition modules to collect the structural dependency information (including network, hardware, and software dependencies) from a variety of sources. With this information, INDaaS then quantifies the independence of systems of interest using pluggable auditing modules, offering various performance, precision, and data secrecy tradeoffs. While the most general and efficient auditing modules assume the auditor is able to obtain all required information, INDaaS can employ private set intersection cardinality protocols to quantify the independence even across businesses unwilling to share their full structural information with anyone. We evaluate the practicality of INDaaS with three case studies via auditing realistic network, hardware, and software dependency structures.

1 Introduction

Cloud services normally require high reliability, and pervasively rely on redundancy techniques to ensure this reliability [7, 10, 12, 29]. Amazon S3, for example, replicates each data object across multiple racks in an S3 region [3]. iCloud rents infrastructures from multiple cloud providers – both Amazon EC2 and Microsoft Azure – for redundancy [28]. Seemingly independent infrastructure components, however, may share deep, hidden dependencies. Failures in these shared dependencies may lead to unexpected *correlated failures*, undermining redundancy efforts [19, 27, 34, 44, 47, 74, 75].

In redundant systems, a *risk group* [35] or RG is a set of components whose simultaneous failures could cause a service outage. Suppose some service *A* replicates critical state across independent servers *B*, *C* and *D* located in three separate racks. The intent of this 3-way redundancy configuration is for all RGs to be of size three, *i.e.*, three servers must fail simultaneously to cause an outage. Unbeknownst to the service provider, however, the three racks share an infrastructure component, such as an aggregation switch *S*. If the switch *S* fails for whatever reason, *B*, *C* and *D* could become unavailable at the same time, causing the service *A* to fail. We say such common dependency introduces an *unexpected RG*, defined as a smaller than expected RG, whose failure could disable the whole service despite redundancy efforts.

This example, while simplistic, nevertheless illustrates documented failures. In an Amazon AWS event [4], a glitch on one Amazon Elastic Block Store (EBS) server disabled the EBS service across Amazon's US-East availability zones. The failure of the EBS service caused correlated failures across multiple Elastic Compute Cloud (EC2) instances utilizing that EBS for storage, and in turn disabled applications designed for redundancy across these EC2 instances. The EBS server in this example was a single common dependency that undermined the EC2's redundancy efforts.

Discovering unexpected common dependencies is extremely challenging [20, 22]. Many diagnostic and forensic approaches attempt to localize or tolerate such failures after they occur [5, 12, 15, 24–27, 31, 37, 43]. These retroactive approaches, however, still require human intervention, leading to prolonged failure recovery time [68]. Google has estimated that "close to 37% of failures are truly correlated" within its global storage system, but they lack the tools to identify these failure correlations systematically [20].

Worse, correlated failures can be hidden not just by inadequate tools or analysts within *one* cloud provider, but also by non-transparent business contracts between cloud providers forming complex multi-level service stacks [19]. Application-level cloud services such as iCloud [28] often redundantly rely on *multiple* cloud providers, *e.g.*, Amazon EC2 and Microsoft Azure. However, a storm in Dublin recently took down a local power source and its backup generator, disabling *both* the Amazon and Microsoft clouds in that region for hours [16]. Providers of higher-level cloud services cannot readily know how independent the lower-level services they build on redundantly really are, since the relevant common dependencies (*e.g.*, power sources) are often propri-

etary internal information, which cloud providers do not normally share [19, 69, 74].

We propose *Independence-as-a-Service* or INDaaS, a novel architecture that aims to address the above problems proactively. Rather than localizing and tolerating failures after an outage, INDaaS collects and audits structural dependency data to evaluate the independence of redundant systems before failures occur. In particular, INDaaS consists of a pluggable set of *dependency acquisition modules* that collect dependency data, and an auditing agent that employs a similarly pluggable set of *auditing modules* to quantify the independence of redundant systems and identify common dependencies that may introduce unexpected correlated failures.

In the dependency acquisition phase, we introduce a uniform representation for different types of dependency data, enabling dependency acquisition modules to be tailored and reused for a particular cloud provider's infrastructure. As an example, our experimental prototype was able to collect dependency data from various sources with respect to network topologies, hardware components, and software packages.

To represent this collected dependency data, INDaaS builds on the traditional fault analysis techniques [52, 60], and further adapts these techniques to audit the independence of redundant systems. Our fault graph representation supports three levels of detail appropriate in different situations: *component-sets*, *fault-sets*, and *fault graphs*. INDaaS can use component-sets to identify shared components even if no failure likelihood information is available. With fault-sets, INDaaS can take failure likelihood information into account. Fault graphs further enable INDaaS to account for deep internal structures involving multiple levels of redundancy.

In its auditing phase, INDaaS offers multiple auditing modules to address tradeoffs among performance, precision, and data secrecy. Our most powerful and informative auditing methods assume that a single independent auditing agent is able to obtain all the required structural dependency data in the clear. This assumption may hold if the agent is a system run by and within a single cloud provider, or if the agent is run by a trusted third party such as a cloud insurance company or a non-profit underwriting agency.

To support independence auditing even across mutually distrustful cloud providers who may be unwilling to share full dependency data with anyone, INDaaS offers *private independence auditing* or PIA. We have explored two approaches to PIA. The first uses secure multi-party computation [72], which offers the best generality in principle but performs adequately only on small dependency datasets [69]. We therefore focus here on the second approach, based on private set intersection cardinality [38, 58]. This approach restricts INDaaS's auditing to the component-set level of detail, but we find it to be practical and scalable to large datasets.

We have developed a prototype INDaaS auditing system, and evaluated its performance with three small but realistic case studies. These case studies exercise INDaaS's two capabilities: 1) proactively quantifying the independence of given redundancy configurations, and 2) identifying potential correlated failures. We find that the prototype scales well. For example, the prototype can audit a cloud dependency structure containing $27,648$ servers and $2,880$ switches/routers, and identify about 90% of relevant dependencies, within 3 hours.

Our INDaaS prototype has many limitations, and would need to be refined and customized to particular cloud environments before real-world deployment. Nevertheless, even as a proof-of-concept, we feel that INDaaS represents one step towards building reliable cloud infrastructures whose redundancy structures can avoid various types of unexpected common-mode failures [23], emergent risks due to overwhelming complexity [44], and proprietary information barriers that naturally arise in the cloud ecosystem [19].

In summary, this paper's contributions are: 1) the first architecture designed to audit the independence of redundant cloud systems before or during deployment; 2) adaptation of fault graph analysis techniques to support multiple levels of detail in explicit dependency structures; 3) an efficient fault graph analysis technique that scales to large datasets representing realistic cloud infrastructures; 4) an application of private set intersection cardinality techniques to enable efficient private independence auditing; 5) a prototype implementation and evaluation of INDaaS's practicality with small but realistic case studies and larger-scale simulations.

2 Architecture Overview

We now present a high-level overview of the INDaaS architecture, deferring details to subsequent sections. Figure 1 illustrates the basic INDaaS workflow, which involves three main roles or types of entities: auditing client, dependency data source, and auditing agent.

The *auditing client*, *i.e.*, Alice in Figure 1, requests an audit of the independence of two or more cloud systems, which may either be operated by Alice herself or rented from other cloud providers, and which she believes to be independent so as to offer redundancy. For example, Alice may request a one-time independence audit prior to deploying a new service onto multiple redundant clouds, like iCloud's use of both Amazon EC2 and Microsoft Azure [28]. Alice might also request periodic audits on a deployed configuration to identify correlated failure risks that configuration changes or evolution might introduce.

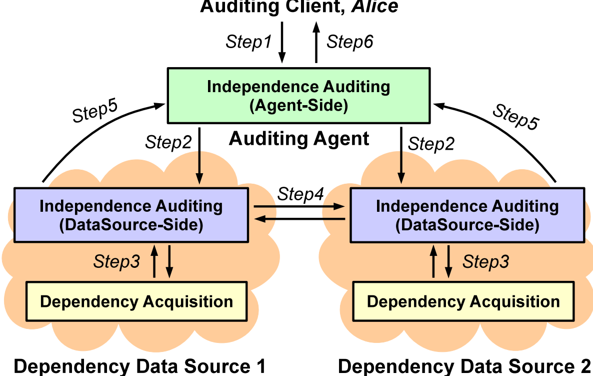


Figure 1: An example INDaaS auditing process, where an auditing client Alice wishes to audit the independence of a two-way redundancy deployment.

Dependency data sources (or data sources for brevity) represent the providers of cloud systems whose independence the auditing client wishes to check. The data sources in practice may be providers of computation, storage and networking components to be used redundantly by the auditing client. INDaaS might be deployed so as to utilize data sources either from a single provider or across multiple providers. In the first case, a storage service like Amazon S3 might provide data sources for each of multiple Amazon data centers offering intra-provider redundancy for S3. In the second, inter-provider scenario, Amazon EC2 and Microsoft Azure might serve as distinct data sources for redundant services rented by iCloud. Either way, as shown in Figure 1, each data source employs pluggable *dependency acquisition* modules to collect structural dependency data on its components such as network topology, hardware devices, or even software packages whose dependencies could lead to common-mode failures (*e.g.*, Heartbleed [23]).

The *auditing agent* mediates the interaction between the auditing client and the data sources. In the case where the auditing agent can obtain the dependency data from all the relevant data sources, the auditing agent constructs a dependency graph based on the data from these data sources. Then, the agent processes the dependency graph and quantifies its independence, or identifies any unexpected common dependencies using a set of pluggable *independence auditing* modules. In the case of private independence auditing, the agent cannot obtain the full dependency data from data sources in cleartext, but supervises a private set intersection cardinality protocol performed by the data sources collaboratively.

We briefly summarize the independence auditing process as illustrated in Figure 1:

Step 1: The auditing client, Alice, specifies to the auditing agent what services she wishes to audit and in what way. This specification includes: a) the relevant data sources; b) the level of redundancy desired; c) the types of components and dependencies to be considered; and d) the metrics used to quantify independence.

Step 2: The auditing agent issues a request to each data source Alice specified.

Step 3: Each specified data source uses one or more dependency acquisition modules to collect the dependency data for future independence auditing (see §3).

Step 4: In the private independence auditing (or PIA) case, the data sources collaborate to obtain the auditing results without revealing the proprietary dependency data to each other (see §4.2).

Step 5: Each data source returns to the auditing agent either the full dependency data for structural independence auditing (see §4.1), or in the PIA case, returns the collaboratively computed independence auditing results.

Step 6: The auditing agent returns to Alice an auditing report quantifying the independence of various redundancy deployments, optionally computing some useful information such as the estimates of correlated failure probabilities and ranked lists of potential risk groups.

Table 1: Format definition of various dependencies.

Type	Dependency Expression
Network	`<src="S" dst="D" route="x,y,z"/>`
Hardware	`<hw="H" type="T" dep="x"/>`
Software	`<pgm="S" hw="H" dep="x,y,z"/>`

3 Dependency Acquisition

Acquiring accurate structural dependency data within heterogeneous cloud systems is non-trivial, and realistic solutions would need to be adapted to different cloud environments. As many dependency acquisition tools have been deployed in today's clouds for various purposes (*e.g.*, system diagnosis) [2, 5, 6, 14, 15, 18, 31, 36, 37, 39], we expect such tools can be adapted and reused to collect the dependency data required by INDaaS.

Towards this end, INDaaS leverages pluggable dependency acquisition modules (DAM), and maintains a uniform representation of different types of dependency data. Different data sources first collect dependency data through their dependency acquisition systems or service monitoring systems, and then adapt the collected data to a common XML-based format illustrated in Table 1. Finally, the DAM stores the adapted dependency data in a database, DepDB, for further processing.

Table 1 shows how our prototype expresses network, hardware, and software dependencies. Each such dependency corresponds to one of the three most common causes of correlated failures [22, 68]: incorrect network

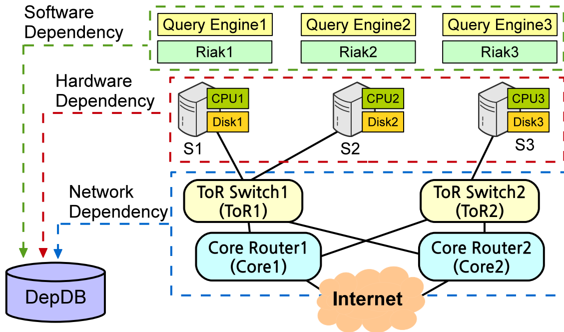


Figure 2: A sample distributed storage system.

```
Network dependencies of S1 and S2:
<src="S1" dst="Internet" route="ToR1,Core1"/>
<src="S1" dst="Internet" route="ToR1,Core2"/>
<src="S2" dst="Internet" route="ToR1,Core1"/>
<src="S2" dst="Internet" route="ToR1,Core2"/>
-------------------------------------
Hardware dependencies of S1 and S2:
<hw="S1" type="CPU" dep="S1-Intel(R)X5550@2.6GHz"/>
<hw="S1" type="Disk" dep="S1-SED900"/>
<hw="S2" type="CPU" dep="S2-Intel(R)X5550@2.6GHz"/>
<hw="S2" type="Disk" dep="S2-SED900"/>
-------------------------------------
Software dependencies of S1 and S2:
<pgm="QueryEngine1" hw="S1" dep="libc6,libgccl">
<pgm="Riak1" hw="S1" dep="libc6,libsvn1">
<pgm="QueryEngine2" hw="S2" dep="libc6,libgccl">
<pgm="Riak2" hw="S2" dep="libc6,libsvn1">
```

Figure 3: A sample of the collected dependency data.

configurations, faulty hardware components, and buggy or insecure software packages.

A *network dependency* describes a route from source `S` to destination `D` via various network components in between, such as routers and/or switches `x`, `y`, and `z`.

A *hardware dependency* describes a physical component, *e.g.*, a disk or CPU of a server. The `hw` field denotes a physical component, and `type` specifies the type of this component such as CPU, disk, RAM, etc. The `dep` field specifies the model number of the component.

A *software dependency* describes the package information of a software component. The `pgm` field denotes the software component `S` itself, `hw` specifies the hardware `H` on which the `S` runs, and `dep` specifies various packages `x`, `y` and `z` used by `S`.

Dependency acquisition examples. Our INDaaS prototype currently includes three dependency acquisition modules employing existing tools to collect various raw dependency data, then adapt them into the common format as discussed above. In particular, we employ NSDMiner [31, 46] to collect network dependencies, HardwareLister [61] to collect hardware dependencies, and apt-rdepends [17] to collect software dependencies. These first-cut INDaaS modules are in no way intended to be definitive but merely aim to provide some examples of realistic dependency acquisition methods.

NSDMiner is a traffic-based network data collector, which discovers network dependencies by analyzing network traffic flows collected from network devices or individual packets [31,46]. HardwareLister (lshw) extracts a target machine's detailed hardware configuration including CPUs, disks and drivers [61]. The apt-rdepends tool extracts the software package and library dependencies for popular Linux software distributions [17].

Figure 2 illustrates a sample distributed storage system. Suppose an auditing client desires two-way redundancy for her service running on two of the three servers S1-S3 within her cloud. She submits to the auditing agent a specification indicating: 1) IP addresses of the three servers, and 2) relevant software components running on these servers. Our current prototype requires the auditing client to list software components of interest manually – *e.g.*, Query Engine and Riak [8] (a distributed database) in this example. With this specification, the auditing agent invokes the dependency acquisition modules (*i.e.*, NSDMiner, lshw, and apt-rdepends) on each server to collect the network, hardware, and software dependencies, and store them in the DepDB, as shown in Figure 3.

4 Independence Auditing

After dependency data acquisition, INDaaS performs independence auditing to generate auditing reports.

As described in §2, INDaaS supports two scenarios. We first present a *structural independence auditing* protocol in §4.1, which assumes data sources are willing to provide the auditing agent with the full dependency data, *e.g.*, for auditing a common cloud provider. We later present a *private independence auditing* protocol in §4.2 to support analysis across multiple cloud providers unwilling to reveal the full dependency data to anyone.

4.1 Structural Independence Auditing

Upon acquiring full dependency data from the data sources, the auditing agent executes our structural independence auditing (SIA) protocol to generate the dependency graph, determine the risk groups, rank the risk groups, and eventually generate an auditing report.

4.1.1 Generating Dependency Graph

To implement structural independence auditing, the auditing agent first generates an explicit dependency graph representation, which will later be used by the pluggable auditing modules. In designing this representation, we adapt traditional fault tree models [52, 60] to a directed

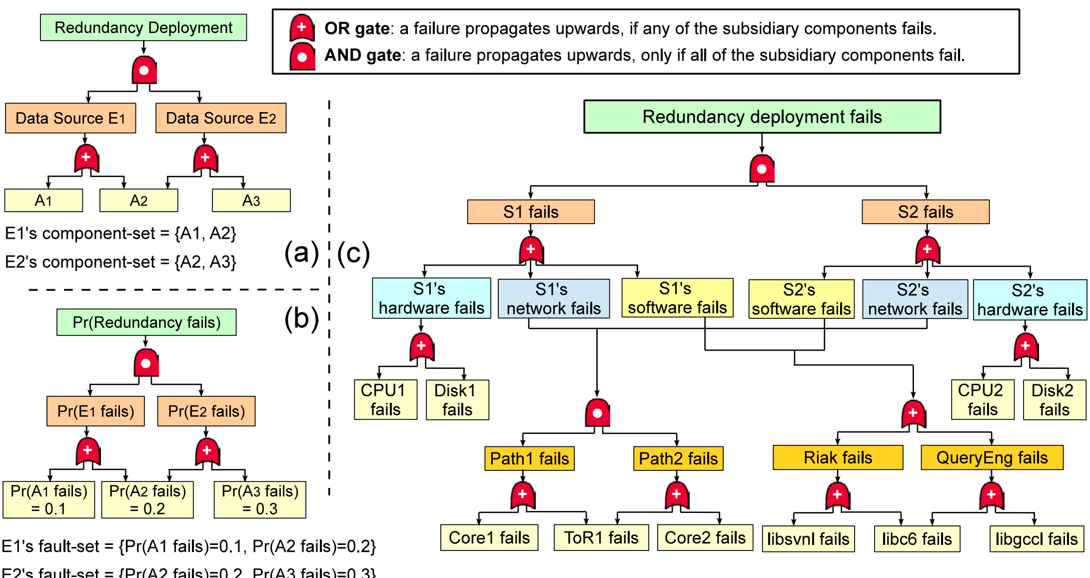


Figure 4: Dependency graphs represented at three different levels of detail: (a) component-set level of detail, (b) fault-set level of detail, and (c) fault graph level of detail.

acyclic graph structure, and generalize the representation to express dependencies at any of three different levels of detail: *component-set*, *fault-set* and *fault graph*.

Component-set. At the most basic level of detail, we organize dependencies in terms of component-sets. As shown in Figure 4(a), if a system E_1 depends on components A_1 and A_2, and another system E_2 depends on components A_2 and A_3, then the two relevant component-sets are $\{A_1,A_2\}$ and $\{A_2,A_3\}$, respectively. E_1 and E_2 are the data sources. At this level of detail, for independence reasoning, we focus only on the presence of shared components – *e.g.*, A_2 – that may lead to correlated failures.

As Figure 4(a) illustrates, we express component-sets in a two-level "*AND*-of-*OR*s" dependency graph. This structure consists of two types of nodes: *components* and *logic gates*. If a component fails (or not), it outputs a 1 (or 0) to its higher-layer logic gate. The two types of logic gates, *AND* and *OR*, depict the different logical relationships among components' failures. For an *OR* gate, if any of its subsidiary components fails, this failure propagates upwards. For an *AND* gate, only if *all* of its subsidiary components fail, the gate propagates a failure upwards. The top-level *AND* gate thus represents redundancy across the data sources (*e.g.*, E_1 and E_2), each of which uses an *OR* gate to connect all its dependent components. Our representation also supports n-of-m redundant deployments ($n \leq m$) via n-of-m *AND* gates.

Fault-set. At the fault-set level of detail, we additionally assign some form of *weight* to each component, *e.g.*, probability of failure over some time period. As shown in Figure 4(b), the failure of A_1 or A_2 leads to the outage of system E_1; thus, the two failure events $\{A_1$ fails, A_2 fails$\}$ form a fault-set. Hereafter, when reasoning at the fault-set level, we assign each failure event a failure probability between 0 and 1. Approaches to obtaining realistic failure probabilities are discussed later in §5.1.

Fault graph. The component-set and fault-set levels of detail assume a single level of redundancy across data sources (*e.g.*, E_1 and E_2), each depending on a "flat" set of components among which any failure causes the respective data source to fail. The fault graph, the richest level of detail INDaaS supports, can describe more complex dependency structures as shown in Figure 4(c). In a fault graph, event nodes having no child nodes are called *basic events*, the root node is called the *top event*, and the remaining nodes are *intermediate events*. Each node in a fault graph has a weight expressing the failure probability of the associated event. A fault graph is evaluated from basic events to the top event. Each top and intermediate event has an *input gate* connecting the lower-layer events. For example, in Figure 4(c), the top event's input gate is an *AND* gate representing top-level redundancy, but the fault graph also expresses internal redundancy via the internal *AND* gates at lower levels.

Building the dependency graph. Any dependency graph, at whichever level of detail, in principle represents the underlying structure of a top-level service across a number of redundant systems. Each such system is a data source where the auditing agent can obtain the dependency data. Automatically building a fault graph with the

dependency data is non-trivial in practice. We summarize here how the auditing agent builds a dependency graph at the fault graph level of detail from top to bottom.

Step 1: The fault graph's top event is the failure of the entire redundancy deployment R.

Step 2: According to the auditing client's specification (see Step 1 in §2), the auditing agent sends a query to the dependency information database DepDB for information about all servers given in the specification. Each server's failure event then becomes a child node of the top event, and an *AND* gate connects the top event with its child nodes to express the servers' redundancy.

Step 3: The auditing agent then queries DepDB for each server's network, hardware, and software dependencies. As a result, each server's failure event has three child nodes, *i.e.*, network, software, and hardware failure events. An *OR* gate connects the server failure event with its three child nodes, since the failure of any of these dependencies effectively causes the server to fail.

Step 4: For the hardware failure event of each server, the auditing agent gets its dependency data from DepDB, then uses an *OR* gate to connect the hardware failure event with its dependencies' failure events.

Step 5: For each server's network failure event, the auditing agent queries DepDB for network paths relevant to the server, then connects them as child nodes to the server's network failure event. The agent puts an *AND* gate between the network failure event and child nodes representing redundant paths, while network devices comprising each path are connected by an *OR* gate.

Step 6: The auditing agent repeats Step 5 to construct the child nodes for each server's software failure event. Different layers of software components are connected by an *OR* gate, and all packages underlying a software component are connected by an *OR* gate.

As an example, the redundancy deployment in Figure 2 may be represented by the fault graph in Figure 4(c). An information-rich fault graph may be "downgraded" to the lower fault-set or component-set levels of detail, by discarding partial information in a fault graph.

Our INDaaS prototype can also compose individual dependency graphs collected from multiple services into more complex aggregate dependency graphs (*e.g.*, EC2 instances depending on services offered by EBS and ELB). Details on dependency graph composition may be found in the associated technical report [75].

4.1.2 Determining Risk Groups

After building a dependency graph, SIA needs to determine risk groups (RGs) of interest in the dependency graph. The SIA provides two pluggable auditing algorithms which make trade-offs between accuracy and efficiency. The *minimal RG* algorithm computes precise results, but its execution time increases exponentially with the size of dependency graph, making it impractical on large datasets. The *failure sampling* algorithm, in contrast, runs much faster but scarifies accuracy. Both algorithms operate on dependency graphs represented at any level of detail. Without loss of generality, hereafter we elaborate on the algorithms at the fault graph level.

Minimal RGs. An RG within a dependency graph is a group of basic failure events with the property that if all of them occur simultaneously, then the top event occurs as well. For example, in Figure 4(a), if A_1 and A_3 fail simultaneously, the redundancy deployment fails. Here, $\{A_1,A_3\}$, $\{A_1,A_2\}$, $\{A_1,A_2,A_3\}$, $\{A_2\}$, and $\{A_2,A_3\}$ are five RGs. Some RGs, however, are more critical than others. We define an RG as a *minimal RG* if the removal of any of its constituent failure events makes it no longer an RG. Consider the following two RGs: $\{A_1,A_2\}$ and $\{A_2,A_3\}$ in Figure 4(a). Neither are minimal RGs because $\{A_2\}$ alone is sufficient to cause the top event to occur; thus, the minimal RGs should be $\{A_2\}$ and $\{A_1,A_3\}$. As another example, the minimal RGs in Figure 4(c) are {ToR1 fails}, {Core1 fails, Core2 fails}, etc.

Minimal RG algorithm. The first algorithm for determining RGs is adapted from classic fault tree analysis techniques [52, 60]. This algorithm traverses a dependency graph G in a reverse breadth-first order (from basic events to the top event). Basic events first generate RGs containing only themselves, while non-basic events produce RGs based on their child events' RGs and their input gates. For a non-basic event, if its input gate is an *OR* gate, the RGs of this event include all its child events' RGs; otherwise, if its input gate is an *AND* gate, each RG of this event is an element of the cartesian product among the RGs of its child events. Traversing the dependency graph G generates all the RGs, and in turn all the minimal RGs through simplification procedures. This algorithm produces precise results, but is NP-hard [59].

Failure sampling algorithm. To address the efficiency issue, we developed an RG detection algorithm based on random sampling, which makes a trade-off between accuracy and efficiency. This algorithm uses multiple sampling rounds, each of which performs a breadth-first traversal of the dependency graph G. Within each sampling round, the algorithm assigns either a 1 or a 0 to each basic event of G based on random coin flipping, where 1 represents failure and 0 represents non-failure. Starting from such an assignment, the algorithm assigns 1s and 0s to all non-basic events from bottom to top based on their logic gates and the values of their child events. After each sampling round, the algorithm checks whether the top event fails. If it fails (*i.e.*, its value is 1), then the algorithm generates an RG consisting of all the basic events being assigned a 1 in this sampling round. The al-

gorithm executes a large number of sampling rounds and aggregates the resulting RGs in all rounds. The failure sampling algorithm offers the linear time complexity, but is non-deterministic and cannot guarantee that the resulting RGs it identifies are minimal RGs. This failure sampling algorithm is similar in principle to heuristic SAT algorithms such as ApproxCount [67], and these methods may offer ways to improve INDaaS failure sampling.

4.1.3 Ranking Risk Groups

After determining RGs, we have two algorithms to rank them and generate the RG-ranking list.

Size-based ranking. To rank RGs at the component-set level or at the unweighted fault graph level, we use a simple size-based ranking algorithm which ranks RGs based on the number of components in each RG. While this algorithm cannot distinguish which potential component failures may be more or less likely, identifying RGs with fewer components – especially any of size 1 indicating no redundancy – can point to areas of the system that may warrant closer manual inspection. For example, in Figure 4(c), the RGs {ToR1} and {libc6} are ranked highest since they have the least size.

Failure probability ranking. In cases where the probabilities of failure events can be estimated, we provide a probability-based ranking algorithm to evaluate RGs at the levels of fault-set and weighted fault graph. This algorithm ranks RGs by their relative importance. Here, for a given RG's failure event (say, C), its relative importance, I_C, is computed using the probability of C, $\Pr(C)$, in comparison to the probability of the top event T, $\Pr(T)$: $I_C = \Pr(C)/\Pr(T)$. Specifically, $\Pr(C)$ is the probability that all the events in C occur simultaneously, and $\Pr(T)$ is computed by the inclusion-exclusion principle where the involved sets are all the minimal RGs of T. In Figure 4(b), since the probabilities of events A_1, A_2 and A_3 are 0.1, 0.2 and 0.3, respectively, we have: $\Pr(T) = 0.1 \cdot 0.3 + 0.2 - 0.1 \cdot 0.3 \cdot 0.2 = 0.224$. Therefore, the relative importances of the minimal RGs $\{A_2$ fails$\}$ and $\{A_1$ fails, A_3 fails$\}$ are: $0.2/0.224 = 0.8929$ and $0.03/0.224 = 0.1339$, respectively. As a result, $\{A_2$ fails$\}$ is ranked higher than $\{A_1$ fails, A_3 fails$\}$.

4.1.4 Generating the Auditing Report

Upon getting the RG-ranking lists for all redundancy deployments, SIA computes an independence score for each of them. If the size-based ranking algorithm is used, a given redundancy deployment R's independence score is computed as $indep(R) = \sum_{i=1}^{n} size(c_i)$, where c_i denotes the ith RG in the R's RG-ranking list, and n denotes the number of top RGs in the RG-ranking list used for this independence evaluation. If the failure probability based ranking algorithm is used, a given redundancy deployment R's independence score is then $indep(R) = \sum_{i=1}^{n} I_{c_i}$, where I_{c_i} denotes the relative importance of c_i.

The auditing agent generates an auditing report by ranking all the redundancy deployments based on their independence scores, and finally sends the report back to the auditing client for reference. With the auditing report, the auditing client might for example select the most independent redundancy deployment for her service.

The auditing report can also help an auditing client understand unexpected common dependencies to focus further analysis. In the case of one documented Amazon EC2 outage, for example [4], we speculate that the availability of an INDaaS auditing report might have enabled the operators to notice that a specific EBS server had become a common dependency, and fix it, thus avoiding the outage.

4.2 Private Independence Auditing

We now address the challenge of independence auditing across mutually distrustful data sources, *e.g.*, multiple cloud providers, who may be unwilling to share dependency data with each other or any third-party auditor. To reflect the motivating deployment model, we use the term *cloud providers* instead of data sources when describing the private independence auditing (PIA) protocol.

The most general and direct approach, explored by Xiao et al. [69], is to use secure multi-party computation (SMPC) [72] to compute and reveal overlap among the datasets of multiple cloud providers while keeping the data themselves private. This approach works in theory, but scales poorly in practice due to its inherent complexity. We find SMPC to be impractical currently even for datasets with only a few hundreds of components.

We thus focus henceforth on a more scalable approach built on private set intersection cardinality techniques [21, 38, 58, 73]. This approach sacrifices generality and dependency graph expressiveness, operating only at the component-set level of detail. The basic idea is to evaluate Jaccard similarity [32] using a private set intersection cardinality protocol [58] to quantify the independence of redundancy configurations.

4.2.1 Trust Assumptions

As described in §2, our architecture consists of entities filling three roles: auditing client, cloud providers (*i.e.*, data sources in Figure 1), and auditing agent.

We assume that auditing clients are potentially malicious and wish to learn as much as possible about the cloud providers' private dependency data. We assume cloud providers and the auditing agent are honest but curious: they run the specified PIA protocol faithfully but

may try to learn additional information in doing so. We assume there is no collusion among cloud providers and the auditing agent. We discuss some potential solutions to dealing with dishonest parties in §5.2.

4.2.2 Technical Building Blocks

There are three technical building blocks that we utilize throughout the PIA design.

Jaccard similarity. Jaccard similarity [32] is a widely-adopted metric for measuring similarity across multiple datasets. Jaccard similarity is defined as $J(S_0, \cdots, S_{k-1}) = |S_0 \cap \cdots \cap S_{k-1}|/|S_0 \cup \cdots \cup S_{k-1}|$ where S_i denotes the ith dataset. A Jaccard similarity J close to 1 indicates high similarity, whereas a J close to 0 indicates the datasets are almost disjoint. In practice, datasets with similarity $J \geq 0.75$ are considered significantly correlated [62]. While there are many other similarity metrics, *e.g.*, the Sørensen-Dice index [57], we choose Jaccard similarity because it is efficient, easy to understand, and extends readily to more than two datasets.

MinHash. Computing the Jaccard similarity incurs a complexity linear with the dataset sizes. In the presence of large datasets, an approximation of the Jaccard similarity based on MinHash is often preferred [11]. The MinHash technique [13] extracts a vector $\{h^{(i)}_{min}(S)\}^{m}_{i=1}$ of a dataset S through deterministic sampling, where $h^{(1)}(\cdot), \cdots, h^{(m)}(\cdot)$ denote m different hash functions, and $h^{(i)}_{min}(S)$ denotes the item $e \in S$ with the minimum value $h^{(i)}(e)$. Let δ denote the number of datasets satisfying $h^{(i)}_{min}(S_0) = \cdots = h^{(i)}_{min}(S_{k-1})$. Then, the Jaccard similarity $J(S_0, \cdots, S_{k-1})$ can be approximated as δ/m. Here, the parameter m correlates to the expected error to the precise Jaccard similarity — a larger m (*i.e.*, more hash functions) yields a smaller approximation error. Broder [13] proves that the expected error of MinHash-based Jaccard similarity estimation is $O(1/\sqrt{m})$.

Private set intersection cardinality. A private set intersection cardinality protocol allows a group of $k \geq 2$ parties, each with a local dataset S_i, to compute the number of overlapping elements among them privately without learning any elements in other parties' datasets. We adopt P-SOP, a private set intersection cardinality protocol based on commutative encryption [58]. In P-SOP, all parties form a logical ring, and agree on the same deterministic hash function (*e.g.*, SHA-1 or MD5). In addition, each party has its own permutation function used to shuffle the elements in its local dataset, as well as its own public/private key pair used for commutative encryption [50, 56]. Commutative encryption offers the property that $E_K(E_J(M)) = E_J(E_K(M))$ where E_X denotes using X's public key to encrypt the message M.

In P-SOP, each party first makes every element in its own dataset S_i identical. Specifically, any element e appearing t times in S_i is represented as t unique elements $\{e\|1, \cdots, e\|t\}$, with '$\|$' being a concatenation operator. Each party then hashes and encrypts every individual element in its dataset, and randomly permutes all the encrypted elements. Afterwards, each party sends the encrypted and permuted dataset to its successor in the ring. Next, once the successor receives the dataset, it simply encrypts each individual element in the received dataset, permutes them, and sends the resulting dataset to its successor. The process repeats until each party receives its own dataset whose individual elements have been encrypted and permuted by all parties in the ring. Finally, all parties share their respective encrypted and permuted datasets, so that they can count the number of common elements in these datasets, *i.e.*, $|\cap_i S_i|$, as well as the number of unique elements in these datasets $|\cup_i S_i|$.

4.2.3 Generating Dependency Graph

To perform private independence auditing, each cloud provider p_i (holding an individual data source) within a given redundancy deployment R first generates its local dependency graph at the component-set level. In addition, each p_i needs to *normalize* its generated component-set. This normalization ensures that the same component shared across different cloud providers always has the same identifier.

Common sources of correlated failures across cloud providers are third-party components such as routers and software packages [19]. Therefore, our current PIA prototype normalizes two types of components: 1) third-party routing elements (*e.g.*, ISP routers), and 2) third-party software packages (*e.g.*, the widely-used OpenSSL toolkit). PIA normalizes these components as follows: 1) for routers, PIA uses their accessible IP addresses as unique identifiers, and 2) for software packages, PIA uses their standard names plus version numbers as unique identifiers. In so doing, any given component in all cloud providers' generated component-sets has a unique normalized identifier.

4.2.4 Auditing Independence Privately

If cloud providers involved in a potential redundancy deployment have relatively small component-sets, PIA takes these (normalized) component-sets S_i directly as input to the private set intersection cardinality protocol (P-SOP) to compute the number of common components $|\cap_i S_i|$ and the number of unique components $|\cup_i S_i|$ across cloud providers. With the two numbers, PIA can compute the Jaccard similarity as $|\cap_i S_i|/|\cup_i S_i|$ to evaluate the independence of this redundancy deployment.

Otherwise, if cloud providers in a potential redundancy deployment have large component-sets, PIA uses m hash functions based on the MinHash technique to map each such component-set to a much smaller dataset S_i, and then takes these MinHash-generated datasets as input to the P-SOP as normal to get the number of common components across cloud providers, *i.e.*, $|\cap_i S_i|$. As discussed in §4.2.2, the Jaccard similarity can then be approximated as $|\cap_i S_i|/m$. This MinHash-based approach leads to much higher efficiency but lower accuracy. To increase the accuracy, we can use more hash functions in MinHash. How to make the trade-off between efficiency and accuracy depends on the application domain.

4.2.5 Generating the Auditing Report

In the design as so far, each cloud provider p_i has computed the Jaccard similarities (or estimated Jaccard similarities using MinHash) corresponding to all the redundancy deployments involving p_i. After collecting these Jaccard similarities from all cloud providers, the auditing agent generates an auditing report ranking all the redundancy deployments based on the Jaccard similarities, and finally sends this report to the auditing client. For an n-of-m redundancy deployment ($n \leq m$), the auditing agent needs to obtain the Jaccard similarity across all the n cloud providers and the similarity across all the m cloud providers, then generate the auditing report.

At the client side, since the auditing client receives only a list ranking all potential redundancy deployments, she obtains no proprietary information about the participating cloud providers' internal infrastructures other than the information produced intentionally to describe their degree of independence.

5 Limitations and Practical Issues

This section discusses a few INDaaS's limitations and areas for further exploration, as well as some practical issues regarding INDaaS deployment.

5.1 Limitations and Potential Solutions

Failure probability acquisition. Part of INDaaS's utility depends on the acquisition of accurate failure probability information. Without this, we cannot perform some auditing operations, *e.g.*, dependency graph generation at the fault-set level and failure probability based ranking. Collecting failure probabilities automatically is a challenging problem in practice, however. Gill et al. proposed one approach [22]: they estimate failure probability by dividing the number of components of a given type that have ever failed during a time period, by the total component population of that given type. They successfully provide the failure probabilities of various network devices (*e.g.*, aggregation switches and core routers) during a one year period. Regarding the failure probabilities of software dependencies, the Common Vulnerability Scoring System (or CVSS) [48] can be used to provide vulnerability-related failure probabilities for many software libraries and packages.

Complex dependency acquisition. Our current software dependency collector takes only static software dependency data into account. In practice, many cloud outages have been caused by more tricky bugs within complex cloud software stacks [5,40,47,51]. Collecting such software dependency data would be challenging, and we are not aware of any existing systematic solutions. A potential solution may need to access the logs generated by various cloud components, and their configuration scripts. For example, we might be able to adapt software failure detection techniques based on mining console logs [70]. Joukov et al. [33] developed a tool that discovers static dependencies between Java programs by parsing these programs' code. In addition, traffic-aware optimizations, *e.g.*, the UDS, BDS and ASD mechanisms proposed by Li et al. [41, 42], can greatly reduce the workload of the network dependency acquisition.

5.2 Practical Issues

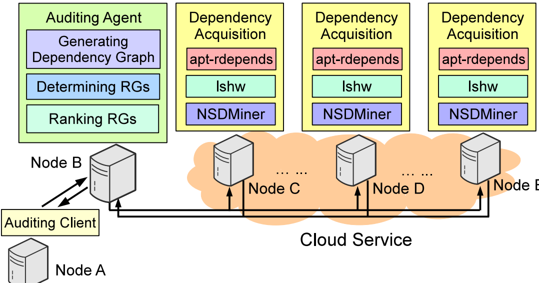
The motivation for auditing clients to use INDaaS is straightforward: they can choose redundancy deployments with better independence property, and can understand unexpected common dependencies which may lead to correlated failures. On the other hand, especially in the PIA case the cloud providers who offer data sources may not explicitly benefit from honestly participating in such a process. We now discuss what incentives the cloud providers have to join PIA, and how they deal with dishonest cloud providers.

Do cloud providers have incentives to join? By participating in PIA, a cloud provider has the opportunity to better understand its potential dependency issues in relation to other cloud providers. While the cloud provider may not learn which specific components overlap with others, it can learn to what extent common dependencies exist between itself and other cloud providers. PIA thus gives cloud providers the opportunity to improve the independence of their deployments. Another potential incentive is that cloud providers not participating in PIA will not appear among the alternative cloud providers that PIA offers to auditing clients. As a result, the clients may be less likely to learn or consider these non-participating alternatives while evaluating various redundancy deployments. These non-participating cloud providers may lose potential customers due to the

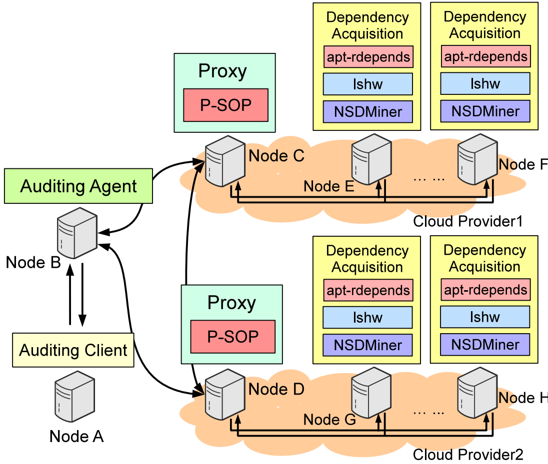
lack of the PIA "reliability label" or merely due to not being on the PIA "certified provider list". Finally, PIA offers cloud providers the opportunity to improve their reputation for transparency and reliability, without risking significant leaks of proprietary secrets about their infrastructure. Joining PIA offers cloud providers a privacy-preserving way to increase the effective transparency of their infrastructures.

Will cloud providers behave honestly? Some cloud providers might execute PIA dishonestly, for example, by declaring a subset of their actual component-sets. In doing so, these providers might benefit from their dishonesty by appearing to have a smaller set intersection and hence greater independence than other providers. Thus, dishonest cloud providers might be ranked higher in the resulting ranking list. To address this issue, we could use the trusted hardware (*e.g.*, TPM) to remotely attest whether cloud providers are performing PIA as required. Recent efforts such as Excalibur [53] have deployed TPM into some cloud platforms successfully.

A less technical solution is to rely on the common business practice of "trust but leave an audit trail." For most executions of PIA, the auditing client simply trusts the participating cloud providers to feed honest and accurate information into the protocol, but the providers must also save and digitally sign the data they used. If an auditing client suspects dishonesty, or during occasional "spot-checks," a specially-authorized independent authority – analogous to the IRS – might perform a "meta-audit" of the provider's PIA records, so that a persistently dishonest participant risks eventually getting caught.



(a) Structural Independence Auditing (SIA).



(b) Private Independence Auditing (PIA).

Figure 5: INDaaS implementation and deployment.

6 Implementation and Evaluation

This section first describes our INDaaS prototype implementation (§6.1), then evaluates its practicality (§6.2) and performance (§6.3).

6.1 Implementation and Deployment

We have built an INDaaS prototype system written in a mix of Python and Java. For clarity, this section focuses first on our implementation of SIA, followed by PIA.

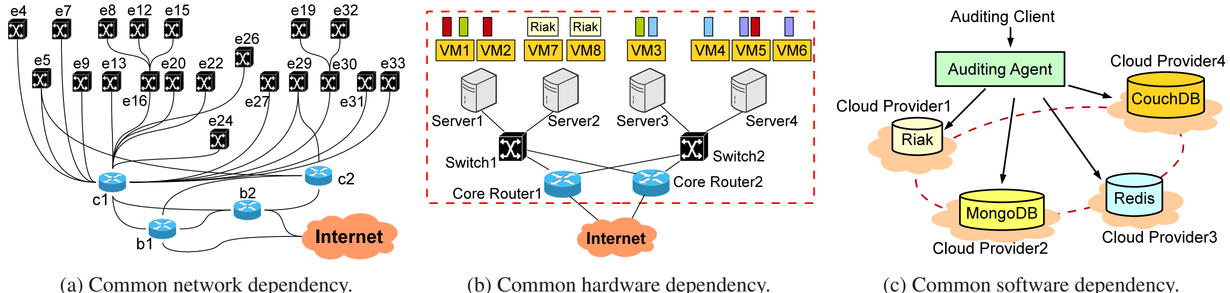
6.1.1 Structural Independence Auditing

Figure 5a shows the key components of an INDaaS prototype in the SIA scenario.

Auditing client. Our auditing client software, currently written in Python, is deployed on a machine maintained by the cloud provider itself, *e.g.*, Node A in Figure 5a. The auditing client communicates with the auditing agent to send the specification and receive the auditing report.

Dependency acquisition. The dependency acquisition modules, written in Python, are deployed on each worker machine to support the audited redundancy deployment in a cloud, *e.g.*, Node C-E in Figure 5a. Our current dependency acquisition implementation includes three open-source tools: NSDMiner [46], lshw [61], and apt-rdepends [17], which are used to collect network, hardware, and software dependencies, respectively. Since each worker machine executes its local dependency acquisition modules separately, the dependency acquisition process can be parallelized.

Auditing agent. The auditing agent, written in Python with the NetworkX [49] library, is deployed on another machine, *e.g.*, Node B in Figure 5a. It collects the dependency data from the dependency acquisition modules on each worker machine over the SSH channels. The agent then audits the collected dependency data, and returns the auditing report back to the auditing client.



(a) Common network dependency. (b) Common hardware dependency. (c) Common software dependency.

Figure 6: Practicality evaluation through three case studies: (a) common network dependency, (b) common hardware dependency, and (c) common software dependency.

6.1.2 Private Independence Auditing

Figure 5b presents the key components of our INDaaS prototype in the PIA scenario.

Auditing client and auditing agent. In PIA, the auditing client and auditing agent are implemented and deployed in a similar way as in SIA, except that the auditing agent is deployed on a machine maintained by a third-party auditor, *i.e.*, not by any audited cloud provider.

Dependency acquisition and proxy. For each cloud provider, there are three dependency acquisition modules deployed on each of its worker machines, as in SIA. Moreover, we implemented a proxy in Java for each cloud provider. The proxy first collects dependency data from the dependency acquisition modules deployed in its own cloud, and then runs the private set intersection cardinality protocol (P-SOP) together with the proxies operated by other cloud providers. In particular, we implemented the P-SOP protocol with MD5, Java permutation, and the commutative RSA encryption scheme [56].

6.2 Practicality Evaluation: Case Studies

This section evaluates INDaaS's practicality through three small but realistic case studies with respect to unexpected common network, hardware, and software dependencies, respectively.

6.2.1 Common Network Dependency

Our first case study targets a scenario similar to the example given in the introduction. A data center operator, Alice, wants to deploy a new service S in her data center, and replicates the critical states of S across two servers within her data center. Before service deployment, Alice uses INDaaS to structurally audit the data center network in order to avoid potential correlated failures resulting from common network dependencies. We used a real data center topology [9] to model Alice's data center network. As shown in Figure 6a, this data center has many Top-of-Rack (ToR) switches (*i.e.*, e1-e33) each of which is connected to an individual rack. There are four core routers (*i.e.*, b1, b2, c1, and c2) connecting ToR switches to the Internet.

The INDaaS first collects network dependencies, and then executes the SIA protocol to provide auditing at the fault graph level. The auditing report generated by our prototype, based on the failure sampling algorithm (which we ran for 10^6 rounds) and the size-based ranking algorithm, suggests that {Rack 5, Rack 29} is the most-independent deployment in this scenario.

A formal analysis indicates that there are 190 different two-way redundancy deployments, among which 27 do not have unexpected RGs. This means, without INDaaS, a random selection leads to only 14% probability for Alice to avoid correlated failures. Furthermore, if we assume the failure probability of all network devices is 0.1, the redundancy deployment {Rack 5, Rack 29} is indeed the one with the lowest failure probability.

6.2.2 Common Hardware Dependency

As shown in Figure 6b, we have built a simple IaaS cloud in the lab with four servers and four switches. We used OpenStack to support the automatic virtual machine (VM) management, and deployed various services on VMs for different uses. In particular, we deployed an S3-like Riak [8] cloud storage service. For redundancy, Riak was run on two VMs (VM7 and VM8).

Before releasing the Riak storage service for public use, we ran SIA to check whether there would be any unexpected RGs. We chose to use the minimal RG algorithm and the size-based ranking algorithm. The top 4 RGs in the RG ranking list generated by our prototype are: {Sever2}, {Switch1}, {Core1 & Core2}, and {VM7 & VM8}. Note that SIA randomly orders RGs with the same size. With this list, we noticed that we had failed to improve the reliability of Riak service via redundant VMs, because the automatic placement module in OpenStack placed the two redundant VMs on the same

Table 2: Ranking lists of two- and three-way redundancy deployments based on Jaccard similarities. Cloud1, 2, 3, and 4 are equipped with Riak, MongoDB, Redis, and CouchDB, respectively.

Rank	Two-Way Redundancy Deployment	Jaccard
1	Cloud2 & Cloud4	0.1419
2	Cloud2 & Cloud3	0.1547
3	Cloud1 & Cloud4	0.2081
4	Cloud1 & Cloud3	0.2939
5	Cloud3 & Cloud4	0.3489
6	Cloud1 & Cloud2	0.5059
Rank	**Three-Way Redundancy Deployment**	**Jaccard**
1	Cloud2 & Cloud3 & Cloud4	0.1128
2	Cloud1 & Cloud2 & Cloud4	0.1207
3	Cloud1 & Cloud3 & Cloud4	0.1353
4	Cloud1 & Cloud2 & Cloud3	0.1536

server (a shared hardware source). As a result, the failure of that server would undermine the redundancy effort. The fundamental cause is that the OpenStack's automatic virtual machine placement policy randomly selects from the least loaded resources to host a VM.

To make the most effective redundancy deployment, we consulted INDaaS for an auditing report on the independence of all potential redundancy deployments. According to the report, which suggests {Server2 and Server3}, we re-deployed the two redundant VMs for the Riak storage service.

6.2.3 Common Software Dependency

The last case study targets a scenario where INDaaS offers private independence auditing across multiple cloud providers. In particular, a service provider, Alice, wants a reliable storage solution leveraging multiple cloud providers, *e.g.*, iCloud uses Amazon EC2 and Microsoft Azure for its reliable storage. Suppose Alice has found four alternative cloud providers: Cloud 1-4, each of which offers a key-value store. Alice then consults INDaaS for a redundancy deployment to avoid correlated failures caused by any shared software dependency [23].

Here, we chose four popular key-value storage systems, *i.e.*, Riak, MongoDB, Redis, and CouchDB. As shown in Figure 6c, we assigned each one to a cloud provider as follows, Cloud1: Riak, Cloud2: MongoDB, Cloud3: Redis, and Cloud4: CouchDB. Suppose each cloud provider has used our prototype to automatically collect the software dependencies of the packages and libraries in its storage system. Our PIA protocol privately computes the Jaccard similarity for each potential redundancy deployment. Table 2 shows the ranking lists of various two- and three-way redundancy deployments.

Table 3: Configurations of the generated topologies.

	Topology A	Topology B	Topology C
# switch ports	16	24	48
# core routers	64	144	576
# agg switches	128	288	1,152
# ToR switches	128	288	1,152
# servers	1,024	3,456	27,648
Total # devices	1,344	4,176	30,528

6.3 Performance Evaluation

We evaluate INDaaS's two major components: SIA and PIA. The performance evaluation was conducted on a research cluster of 40 workstations equipped with Intel Xeon Quad Core HT 3.7 GHz CPU and 16 GB RAM.

6.3.1 SIA: Efficiency v.s. Accuracy

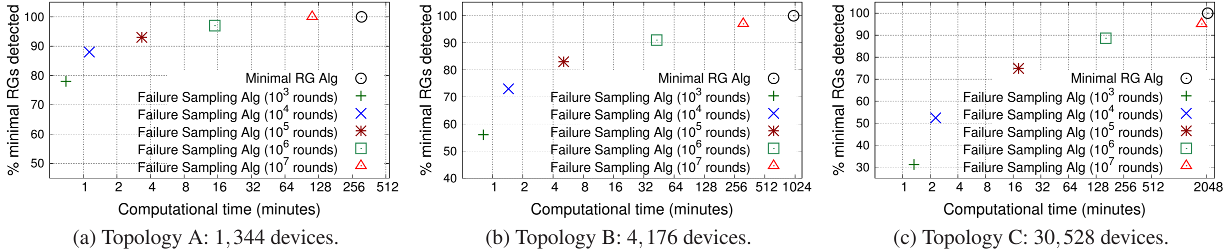
We first explore the efficiency/accuracy trade-off between SIA's two algorithms for analyzing a dependency graph: the minimal RG algorithm and the failure sampling algorithm (see §4.1.2). We generate three topologies from a small-scale cloud deployment to a large-scale deployment, based on the three-stage fat tree model [45]. These topologies include the typical components within a commercial data center: servers, Top-of-Rack (ToR) switches, aggregation switches, and core routers. Table 3 gives the detail of these generated topologies.

We compare the computational overhead of the accurate but NP-hard minimal RG algorithm to that of the failure sampling algorithm with various sampling rounds (10^3 to 10^7). Figure 7 shows the result that the failure sampling algorithm runs much more efficiently than the minimal RG algorithm while achieving a reasonably high accuracy. For example, in topology B, the failure sampling algorithm uses 90 minutes to detect 92% of all the minimal RGs with 10^6 sampling rounds, in comparison to 1046 minutes for the minimal RG algorithm.

6.3.2 PIA: System Overheads

To better understand the performance of PIA, we implemented a comparable private independence auditing system based on another private set intersection cardinality protocol, Kissner and Song (KS) [38], and then compared this system with our PIA system.

For a private independence auditing system, the cryptographic operations tend to be the major computational bottleneck. Thus, we evaluate PIA by comparing PIA's P-SOP protocol with the comparable system's KS protocol. Specifically, the cryptographic primitives of P-SOP are hashing, commutative encryption, and permutation. The KS protocol is mainly built on hashing, homomorphic crypto operations, and permutation.

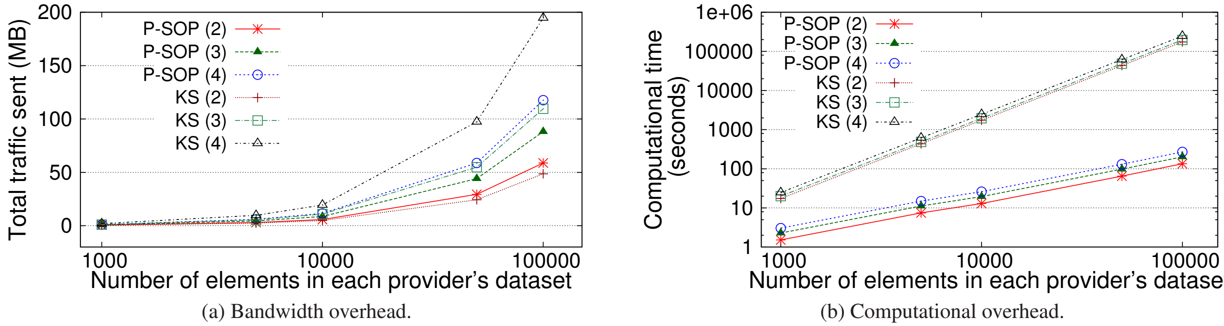


(a) Topology A: 1,344 devices.

(b) Topology B: 4,176 devices.

(c) Topology C: 30,528 devices.

Figure 7: Performance evaluation of the minimal RG algorithm and the failure sampling algorithm in SIA.



(a) Bandwidth overhead.

(b) Computational overhead.

Figure 8: System overhead evaluation of PIA. P-SOP (k) and KS(k) mean that there are k cloud providers participating in the P-SOP and KS protocols, respectively. The commutative encryption in P-SOP uses a 1024-bit key, and the homomorphic encryption in KS also uses a 1024-bit key.

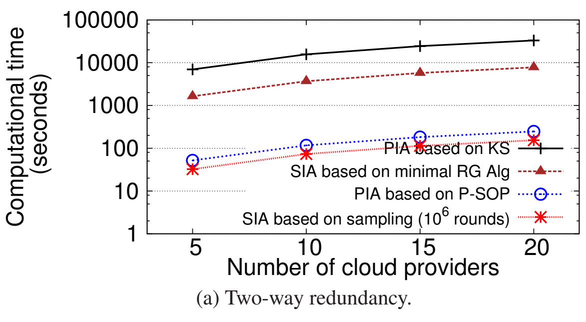
In the evaluation, there are k cloud providers with n elements in each provider's local dataset. We set k to 2, 3 and 4, and vary n between 1,000 and 100,000 to cover a wide range of real-world settings. We measure and compare P-SOP with KS in terms of their bandwidth and computational overheads at each such cloud provider. Figure 8a and 8b show the bandwidth overhead and computational overhead, respectively.

With a small number of cloud providers (*e.g.*, $k = 2$), the bandwidth overhead of KS is comparable to that of P-SOP. However, with an increasing number of cloud providers, KS's bandwidth overhead increases much faster than P-SOP's. With respect to the computational overhead, P-SOP outperforms KS by a few orders of magnitude although both protocols' computational overheads increase almost linearly with the number of elements in each cloud provider's dataset. Altogether, the evaluation shows that our PIA system can efficiently handle large cloud providers each with even hundreds of thousands of system components.

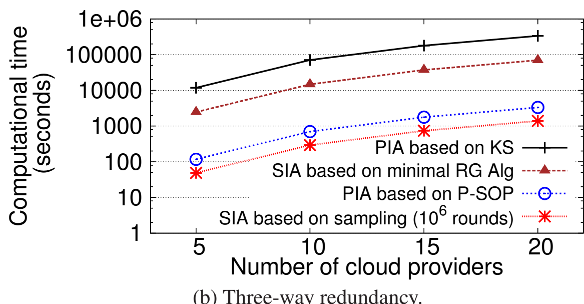
6.3.3 Comparison: SIA Versus PIA

Compared with the SIA where there is a trusted auditor, we would also like to understand how much extra overhead the PIA approach incurs to preserve the secrecy of each participating cloud provider's data. Assume each cloud provider maintains a local dataset containing 10,000 elements. To preserve secrecy for each cloud provider, an auditing client relies on either the PIA system or the comparable KS-based system to determine the most independent redundancy deployment. For a comparison, we also assume another setting where there exists a trusted auditor who knows all cloud providers' datasets. This trusted auditor runs SIA at the component-set level of detail based on the minimal RG algorithm or the failure sampling algorithm with 10^6 rounds.

Figure 9a and 9b show the computational overheads of these independence calculations for all potential two- and three-way redundancies, respectively. As expected, preserving the secrecy of cloud providers' data does incur extra overhead. Surprisingly, this cost is not as high as might be expected: we see that the computational overhead of "PIA based on P-SOP" is less than twice that of "SIA based on sampling (10^6 rounds)". The SIA sampling scheme does implement a more general analysis than PIA, supporting fault graphs rather than just component sets. Unsurprisingly, both "PIA based on KS" and "SIA based on minimal RG Alg" do not scale well.



(a) Two-way redundancy.



(b) Three-way redundancy.

Figure 9: Performance comparison between SIA and PIA. Each cloud provider maintains a 10,000-element dataset.

7 Related Work

Providing audits for clouds is a well-known approach to increase reliability [54]. Practical and systematic cloud auditing, however, still remains an open problem. To the best of our knowledge, INDaaS is the first systematic effort to enable independence audits for cloud services.

Privacy-preserving auditing systems. Following the auditing concept proposed by Shah et al. [54], many privacy-preserving auditing systems have been proposed extending this approach [55, 63–66, 71].

Similar to PIA, iRec [74] and Xiao et al. [69] also focused on analyzing correlated failures resulting from the common infrastructure dependencies across multiple cloud providers. These efforts proposed using the private set intersection cardinality protocol [21] and the secure multi-party computation protocol [72] to perform the dependency analysis in a privacy-preserving fashion, respectively. These initial efforts did not scale to handle realistically large cloud datasets, however,

Diagnosis & accountability systems. Diagnosis systems, unlike auditing, attempt to discover failures after they occur. For example, many inference-based diagnosis systems [5, 15, 31, 37] have been proposed to obtain the network dependencies of a cloud service when a failure occurs. Unlike existing diagnosis systems, NetPilot [68] aimed to mitigate these failures rather than directly localize their sources.

Accountability systems attempt to place blame after failures occur, whereas our auditing system attempts to prevent failures in the first place. Haeberlen [24] proposed using third-party verifiable evidence to determine whether the cloud customer or the cloud provider should be held liability when a failure occurs.

Private set operations. Secure multi-party computation (SMPC) [72] is a general approach to supporting computation on private data including set operations. However, current circuit-based SMPC protocols are too expensive and scale poorly to large computations. Arawal et al. [1] proposed a private set intersection cardinality protocol based on commutative encryption. This protocol was limited to two-party cases, however. Vaidya and Clifton [58] extended this protocol to support more than two parties, and optimized its efficiency.

The first private set intersection cardinality protocol based on homomorphic encryption was proposed by Freedman et al. [21], which could privately compute the number of elements common to two datasets. Hohenberger et al. proposed enhancements to this protocol protocol [30]. Later, Kissner and Song proposed multi-party private set operations based upon homomorphic encryption and polynomial generation [38].

8 Conclusion

This paper has presented INDaaS, an architecture to audit the independence of future or existing redundant service deployments in the cloud. While only a start, our proof-of-concept prototype and experiments suggest that INDaaS could be both practical and effective in detecting and heading off correlated failure risks before they occur.

Acknowledgments

We thank our shepherd, Timothy Roscoe, and the anonymous reviewers for their insightful comments. We also thank Gustavo Alonso, Hongqiang Liu, Jeff Mogul, Ruzica Piskac, Xueyuan Su, Hongda Xiao, and Sebastian Zander for their valuable feedback on earlier drafts of this paper. This research was sponsored by the NSF under grants CNS-1017206 and CNS-1149936.

References

[1] Rakesh Agrawal, Alexandre V. Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *ACM SIGMOD*, June 2003.

[2] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[3] Amazon S3's redundant storage. http://aws.amazon.com/s3/, accessed on Sep 9, 2014.

[4] Amazon Web Services Team. Summary of the October 22, 2012 AWS service event in the US-East region. https://aws.amazon.com/message/680342/, accessed on Sep 9, 2014.

[5] Paramvir Bahl, Ranveer Chandra, Albert G. Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM*, August 2007.

[6] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[7] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic, and Ido Zachevsky. Robust data sharing with key-value stores. In *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2012.

[8] Basho Technologies. Riak. http://basho.com/riak/, accessed on Sep 9, 2014.

[9] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conference (IMC)*, November 2010.

[10] Alysson Neves Bessani, Miguel P. Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, April 2011.

[11] Carlo Blundo, Emiliano de Cristofaro, and Paolo Gasti. EsPRESSo: Efficient privacy-preserving evaluation of sample set similarity. In *DPM/SETOP*, September 2012.

[12] Nicolas Bonvin, Thanasis G. Papaioannou, and Karl Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *ACM Symposium on Cloud Computing (SoCC)*, June 2010.

[13] Andrei Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES)*, June 1997.

[14] Mike Y. Chen, Anthony Accardi, Emre Kiciman, David A. Patterson, Armando Fox, and Eric A. Brewer. Path-based failure and evolution management. In *1st USENIX Symposium on Networked System Design and Implementation (NSDI)*, March 2004.

[15] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2008.

[16] Jack Clark. Lightning strikes Amazon's European cloud. *ZDNet*, August 2011. http://www.zdnet.com/lightning-strikes-amazons-european-cloud-3040093641/, accessed on Sep 9, 2014.

[17] Debian. Package apt-rdepends: Recursively lists package dependencies. http://packages.debian.org/sid/apt-rdepends, accessed on Sep 9, 2014.

[18] John Dunagan, Nicholas J. A. Harvey, Michael B. Jones, Dejan Kostic, Marvin Theimer, and Alec Wolman. FUSE: Lightweight guaranteed distributed failure notification. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[19] Bryan Ford. Icebergs in the clouds: the *other* risks of cloud computing. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2012.

[20] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[21] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, May 2004.

[22] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM*, August 2011.

[23] Dan Greer. Heartbleed as metaphor. *Lawfare*, April 2014. http://www.lawfareblog.com/2014/04/heartbleed-as-metaphor/, accessed on Sep 9, 2014.

[24] Andreas Haeberlen. A case for the accountable cloud. In *3rd ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, October 2009.

[25] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschelnd. Accountable virtual machines. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[26] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[27] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *2nd Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[28] Devindra Hardawar. Apple's iCloud runs on Microsoft's Azure and Amazon's cloud. *VentureBeat News*, September 2011. http://venturebeat.com/2011/09/03/icloud-azure-amazon/, accessed on Sep 9, 2014.

[29] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. Next Step, the Cloud: Understanding modern web service deployment in EC2 and Azure. In *Internet Measurement Conference (IMC)*, October 2013.

[30] Susan Hohenberger and Stephen A. Weis. Honest-verifier private disjointness testing without random oracles. In *6th Workshop on Privacy Enhancing Technologies*, 2006.

[31] Barry Peddycord III, Peng Ning, and Sushil Jajodia. On the accurate identification of network service dependencies in distributed systems. In *26th Large Installation System Administration Conference (LISA)*, December 2012.

[32] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et du Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37(142):547–579, June 1901.

[33] Nikolai Joukov, Vasily Tarasov, Joel Ossher, Birgit Pfitzmann, Sergej Chicherin, Marco Pistoiz, and Takaaki Tateishi. Static discovery and remediation of code-embedded resource dependencies. In *Integrated Network Management*, pages 233–240, 2011.

[34] Flavio Paiva Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker. Surviving Internet catastrophes. In *USENIX Annual Technical Conference*, pages 45–60, April 2005.

[35] Ivan P Kaminow and Thomas L Koch. *Optical Fiber Telecommunications IIIA*. Academic Press, New York, 1997.

[36] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. Shrink: A tool for failure diagnosis in IP networks. In *SIGCOMM MineNet Workshop*, August 2005.

[37] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM*, August 2009.

[38] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *25th Annual International Cryptology Conference (CRYPTO)*, August 2005.

[39] Ramana Rao Kompella, Jennifer Yates, Albert G. Greenberg, and Alex C. Snoeren. IP fault localization via risk modeling. In *2nd USENIX Symposium on Networked System Design and Implementation (NSDI)*, May 2005.

[40] Sarah Kuranda. The 10 Biggest Cloud Outages of 2013. *CRN*, January 2014. http://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm, accessed on Sep 9, 2014.

[41] Zhenhua Li, Cheng Jin, Tianyin Xu, Christo Wilson, Yao Liu, Linsong Cheng, Yunhao Liu, Yafei Dai, and Zhi-Li Zhang. Towards network-level efficiency for cloud storage services. In *14th ACM Internet Measurement Conference (IMC)*, November 2014.

[42] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y. Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient batched synchronization in Dropbox-like cloud storage services. In *14th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, December 2013.

[43] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[44] Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. In *1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, April 2006.

[45] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM*, August 2009.

[46] Arun Natarajan, Peng Ning, Yao Liu, Sushil Jajodia, and Steve E. Hutchinson. NSDMiner: Automated discovery of network service dependencies. In *IEEE INFOCOM*, December 2012.

[47] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[48] National Institute of Standards and Technology. Common Vulnerability Scoring System (CVSS). http://nvd.nist.gov/cvss.cfm, accessed on Sep 9, 2014.

[49] NetworkX. http://networkx.github.com/, accessed on Sep 9, 2014.

[50] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (Corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[51] Rahul Potharaju and Navendu Jain. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *ACM Symposium on Cloud Computing (SoCC)*, October 2013.

[52] Chittoor V. Ramamoorthy, Gary S. Ho, and Yih-Wu Han. Fault tree analysis of computer systems. In *AFIPS National Computer Conference*, 1977.

[53] Nuno Santos, Rodrigo Rodrigues, Krishna P Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *21st USENIX Security Symposium (USENIX Security)*, August 2012.

[54] Mehul A. Shah, Mary Baker, Jeffrey C. Mogul, and Ram Swaminathan. Auditing to keep online storage services honest. In *11th Workshop on Hot Topics in Operating Systems (HotOS)*, May 2007.

[55] Mehul A. Shah, Ram Swaminathan, and Mary Baker. Privacy-preserving audit and extraction of digital contents. Technical Report HPL-2008-32R1, HP Laboratories, April 2008.

[56] Adi Shamir, Ron Rivest, and Leonard Adleman. Mental poker. Technical Report LCS/TM-125, Massachusetts Institute of Technology, February 1979.

[57] T. Sørensen. *A Method of Establishing Groups of Equal Amplitude in Plant Sociology Based on Similarity of Species Content and Its Application to Analyses of the Vegetation on Danish Commons*. I kommission hos E. Munksgaard, 1948.

[58] Jaideep Vaidya and Chris Clifton. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security*, (4):593–622, 2005.

[59] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal of Computing*, 8(3):410–421, 1979.

[60] William E. Vesely, Francine F. Goldberg, Norman H. Roberts, and David F. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, January 1981.

[61] Lyonel Vincent. Hardware Lister (lshw). http://ezix.org/project/wiki/HardwareLiSter, accessed on Sep 9, 2014.

[62] Kevin Walsh and Emin Gün Sirer. Experience with an object reputation system for Peer-to-Peer filesharing. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[63] Cong Wang, Sherman S. M. Chow, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers*, 62(2):362–375, 2013.

[64] Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Toward publicly auditable secure cloud data storage services. *IEEE Network*, 24(4):19–24, 2010.

[65] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *IEEE INFOCOM*, March 2010.

[66] Qian Wang, Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):847–859, 2011.

[67] Wei Wei and Bart Selman. A new approach to model counting. In *8th Internal Conference on Theory and Applications of Satisfiability Testing (SAT)*, June 2005.

[68] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. NetPilot: Automating datacenter network failure mitigation. In *ACM SIGCOMM*, August 2012.

[69] Hongda Xiao, Bryan Ford, and Joan Feigenbaum. Structural cloud audits that protect private information. In *ACM Cloud Computing Security Workshop (CCSW)*, November 2013.

[70] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, October 2009.

[71] Kan Yang and Xiaohua Jia. Data storage auditing service in cloud computing: Challenges, methods and opportunities. *World Wide Web*, 15(4):409–428, 2012.

[72] Andrew Chi-Chih Yao. Protocols for secure computations (Extended abstract). In *23rd Annual Symposium on Foundations of Computer Science (FOCS)*, November 1982.

[73] Sebastian Zander, Lachlan L. H. Andrew, and Grenville Armitage. Scalable private set intersection cardinality for capture-recapture with multiple private datasets. In *Centre for Advanced Internet Architectures, Technical Report 130930A*, 2013.

[74] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. An untold story of redundant clouds: Making your service deployment truly reliable. In *9th Workshop on Hot Topics in Dependable Systems (HotDep)*, November 2013.

[75] Ennan Zhai, David Isaac Wolinsky, Hongda Xiao, Hongqiang Liu, Xueyuan Su, and Bryan Ford. Auditing the structural reliability of the clouds. Technical Report YALEU/DCS/TR-1479, Department of Computer Science, Yale University, 2014. Available at http://cpsc.yale.edu/sites/default/files/files/tr1479.pdf, accessed on Sep 9, 2014.

Characterizing Storage Workloads with Counter Stacks

Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield
Coho Data

Abstract

Existing techniques for identifying working set sizes based on miss ratio curves (MRCs) have large memory overheads which make them impractical for storage workloads. We present a novel data structure, the *counter stack*, which can produce approximate MRCs while using sublinear space. We show how counter stacks can be checkpointed to produce workload representations that are many orders of magnitude smaller than full traces, and we describe techniques for estimating MRCs of arbitrary workload combinations over arbitrary windows in time. Finally, we show how online analysis using counter stacks can provide valuable insight into live workloads.

1 Introduction

Caching is poorly understood. Despite being a pervasive element of computer system design – one that spans processor, storage system, operating system, and even application architecture – the effective sizing of memory tiers and the design of algorithms that place data within them remains an art of characterizing and approximating common case behaviors.

The design of hierarchical memories is complicated by two factors: First, the collection of live workload-specific data that might be analyzed to make "application aware" decisions is generally too expensive to be worthwhile. Approaches that model workloads to make placement decisions risk consuming the computational and memory resources that they are trying to preserve. As a result, systems in many domains have tended to use simple, general purpose algorithms such as LRU to manage cache placement. Second, attempting to perform offline analysis of access patterns suffers from the performance overheads imposed in trace collection, and the practical challenges of both privacy and sheer volume, in sharing and analyzing access traces.

Today, these problems are especially pronounced in designing enterprise storage systems. Flash memories are now available in three considerably different form factors: as SAS or SATA-attached solid state disks, as NVMe devices connected over the PCIe bus, and finally as flash-backed nonvolatile RAM, accessible over a DIMM interface. These three connectivity models all use the same underlying flash memory, but present performance and pricing that are pairwise 1-2 orders of magnitude apart. Further, in addition to solid-state memories, spinning disks remain an economical option for the storage of cold data.

This paper describes an approach to modeling, analyzing, and reasoning about memory access patterns that has been motivated through our experience in designing a hierarchical storage system [10] that combines these varying classes of storage media. The system is a scalable, network-attached storage system that can benefit from workload awareness in two ways: First, the system can manage allocation of the memory hierarchy in response to workload characteristics. Second, the capacity at each level of the hierarchy can be independently expanded to satisfy application demands, by adding additional hardware. Both of these properties require a more precise ability to understand and characterize individual storage workloads, and in particular their working set sizes over time.

Miss ratio curves (MRCs) are an effective tool for assessing working set sizes, but the space and time required to generate them make them impractical for large-scale storage workloads. We present a new data structure, the *counter stack*, which can generate approximate MRCs in sublinear space, for the first time making this type of analysis feasible in the storage domain.

Counter stacks use probabilistic counters [18] to estimate MRCs. The original approach to generating MRCs is based on the observation that a block's 'stack distance' (also known as its 'reuse distance') gives the capacity needed to cache it, and this distance is exactly the number of unique blocks accessed since the previous request for the block. The key idea behind counter stacks is that probabilistic counters can be used to efficiently estimate stack distances, allowing us to compute approximate MRCs at a fraction of the cost of traditional techniques.

Counter stacks are fast. Our Java implementation can process a week-long trace of 13 enterprise servers in 17 minutes using just 80 MB of RAM; at a rate of 2.3 million requests per second, the approach is practical for on-

line analysis in production systems. By comparison, a recent C implementation of a tree-based optimization [27] of Mattson's original stack algorithm [23] takes roughly an hour and 92 GB of RAM to process the same trace.

Our contributions in this paper are threefold. First, we introduce a novel technique for estimating miss ratio curves using counter stacks, and we evaluate the performance and accuracy of this technique. Second, we show how counter stacks can be periodically checkpointed and streamed to disk to provide a highly compressed representation of storage workloads. Counter stack *streams* capture important details that are discarded by statistical aggregation while at the same time requiring orders of magnitude less storage and processing overhead than full request traces; a counter stack stream of the compressed 2.9 GB trace mentioned above consumes just 11 MB. Third, we present techniques for working with multiple independent counter stacks to estimate miss ratio curves for new workload combinations. Our library implements *slice*, *shift*, and *join* operations, enabling the nearly-instantaneous computation of MRCs for arbitrary workload combinations over arbitrary windows in time. These capabilities extend the functionality of MRC analysis and provide valuable insight into live workloads, as we demonstrate with a number of case studies.

2 Background

The many reporting facilities embedded in the modern Linux storage stack [5, 7, 19, 25] are testament to the importance of being able to accurately characterize live workloads. Common characterizations typically fall into one of two categories: coarse-grain aggregate statistics and full request traces. While these representations have their uses, they can be problematic for a number of reasons: averages and histograms discard key temporal information; sampling is vulnerable to the often bursty and irregular nature of storage workloads; and full traces impose impractical storage and processing overheads. New representations are needed which preserve the important features of full traces while remaining manageable to collect, store, and query.

Working set theory [12] provides a useful abstraction for describing workloads more concisely, particularly with respect to how they will behave in hierarchical memory systems. In the original formulation, working sets were defined as the set of all pages accessed by a process over a given epoch. This was later refined by using LRU modelling to derive an MRC for a given workload and restricting the working set to only those pages that exhibit strong locality. Characterizing workloads in terms of the unique, 'hot' pages they access makes it easier to understand their individual hardware requirements, and has proven useful in CPU cache management for many years [21, 28, 35]. These concepts hold for storage workloads as well, but their application in this domain is challenging for two reasons.

First, until now it has been prohibitively expensive to calculate the working set of storage workloads due to their large sizes. Mattson's original stack algorithm [23] required $O(NM)$ time and $O(M)$ space for a trace of N requests and M unique elements. An optimization using a balanced tree to maintain stack distances [1] reduces the time complexity to $O(N \log M)$, and recent approximation techniques [14, 38] reduce the time complexity even further, but they still have $O(M)$ space overheads, making them impractical for storage workloads that may contain billions of unique blocks.

Second, the extended duration of storage workloads leads to subtleties when reasoning about their working sets. CPU workloads are relatively short-lived, and in many cases it is sufficient to consider their working sets over small time intervals (e.g., a scheduling quantum) [42]. Storage workloads, on the other hand, can span weeks or months and can change dramatically over time. MRCs at this scale can be tricky: if they include too little history they may fail to capture important recurring patterns, but if they include too much history they can significantly misrepresent recent behavior.

This phenomenon is further exacerbated by the fact that storage workloads already sit behind a file system cache and thus typically exhibit longer reuse distances than CPU workloads [43]. Consequently, cache misses in storage workloads may have a more pronounced effect on miss ratios than CPU cache misses, because subsequent reaccesses are likely to be absorbed by the file system cache rather than contributing to hits at the storage layer.

One implication of this is that MRC analysis needs to be performed over various time intervals to be effective in the storage domain. A workload's MRC over the past hour may differ dramatically from its MRC over the past day; both data points are useful, but neither provides a complete picture on its own.

This leads naturally to the notion of a *history of locality*: a workload representation which characterizes working sets as they change over time. Ideally, this representation contains enough information to produce MRCs over arbitrary ranges in time, in much the same way that full traces support statistical aggregation over arbitrary intervals. A naïve implementation could produce this representation by periodically instantiating new Mattson stacks at fixed intervals of a trace, thereby modelling independent LRU caches with various amounts of history, but such an ap-

proach would be impractical for real-world workloads.

In the following section we describe a novel technique for computing stack distances (and by extension, MRCs), from an inefficient, idealized form of counter stacks. Section 4 explains several optimizations which allow a practical counter stack implementation that requires sublinear space, and Section 5 presents the additional operations that counter stacks support, such as slicing and joining.

3 Counter Stacks

Counter stacks capture locality properties of a sequence of accesses within an address space. In the context of a storage system, accesses are typically read or write requests to physical disks, logical volumes, or individual files. A counter stack can process a sequence of requests as they occur in a live storage system, or it can process, in a single pass, a trace of a storage workload. The purpose of a counter stack is to represent specific characteristics of the stream of requests in a form that is efficient to compute and store, and that preserves enough information to characterize aspects of the workload, such as cache behaviour.

Rather than representing a trace as a sequence of requests for specific addresses, counter stacks maintain a list of counters, which are periodically instantiated while processing the trace. Each counter records the number of *unique* trace elements observed since the inception of that counter; this captures the size of the working set over the corresponding portion of the trace. Computing and storing samples of working set size, rather than a complete access trace, yields a very compact representation of the trace that nevertheless reveals several useful properties, such as the number of unique blocks requested, or the stack distances of all requests, or phase changes in the working set. These properties enable computation of MRCs over arbitrary portions of the trace. Furthermore, this approach supports composition and extraction operations, such as joining together multiple traces or slicing traces by time, while examining only the compact representation, not the original traces.

3.1 Definition

A counter stack is an in-memory data structure that is updated while processing a trace. At each time step, the counter stack can report a list of values giving the numbers of distinct blocks that were requested between the current time and *all previous* points in time. This data structure evolves over time, and it is convenient to display its history as a matrix, in which each column records the values reported by the counter stack at some point in time.

Formally, given a trace sequence $(e_1 \dots e_N)$, where e_i is the ith trace element, consider an $N \times N$ matrix C whose entry in the ith row and jth column is the number of distinct elements in the set $\{e_i \dots e_j\}$. For example, the trace (a, b, c, a) yields the following matrix.

(	a,	b,	c,	a,	)
	1	2	3	3	
		1	2	3	
			1	2	
				1	

The jth column of this matrix gives the values reported by the counter stack at time step j, i.e., the numbers of distinct blocks that were requested between that time and all previous times. The ith row of the matrix can be viewed as the sequence of values produced by the counter that was instantiated at time step i.

The in-memory counter stack only stores enough information to produce, at any point in time, a single column of the matrix. To compute our desired properties over arbitrary portions of the trace, we need to store the entire history of the data structure, i.e., the entire matrix. However, the history does not need be stored in memory. Instead, at each time step we write to disk the current column of values reported by the counter stack. This can be viewed as checkpointing, or incrementally updating, the on-disk representation of the matrix. This on-disk representation is called a *counter stack stream*; for conciseness we will typically refer to it simply as a *stream*.

3.2 LRU Stack Distances

Stack distances and MRCs have numerous applications in cache sizing [23], memory partitioning between processes or VMs [20,34,35,42], garbage collection frequency [39], program analysis [14,41], workload phase detection [31], etc. A significant obstacle to the widespread use of MRCs is the cost of computing them, particularly the high storage cost [4, 27, 30, 33, 40] – all existing methods require linear space. Counter stacks eliminate this obstacle by providinge xtremely efficient MRC computation while using sublinear space.

In this subsection we explain how stack distances, and hence MRCs, can be derived from counter stack streams. Recall that the stack distance of a given request is the number of distinct elements observed since the last reference to the requested element. Because a counter stack stores information about distinct elements, determining the stack distance is straightforward. At time step j one must find the last position in the trace, i, of the requested element, then examine entry C_{ij} of the matrix to determine the number of distinct elements requested between

times i and j. For example, let us consider the matrix given in Section 3.1. To determine the stack distance for the second reference to trace element a at position 4, whose previous reference was at position 1, we look up the value $C_{1,4}$ and get a stack distance of 3.

This straightforward method ignores a subtlety: how can one find the last position in the trace of the requested element? It turns out that this information is implicitly contained in the counter stack. To explain this, suppose that the counter that was instantiated at time i does not increase during the processing of element e_j. Since this counter reports the number of *distinct* elements that it has seen, we can infer that this counter has already seen element e_j. On the other hand, if the counter instantiated at time $i+1$ does increase while processing e_j, then we can infer that this counter has not yet seen element e_j. Combining those inferences, we can conclude that i is the position of last reference.

These observations lead to a finite-differencing scheme that can pinpoint the positions of last reference. At each time step, we must determine how much each counter increases during the processing of the current element of the trace. This is called the *intra-counter* change, and it is defined to be

$$\Delta x_{ij} \;=\; C_{i,j} - C_{i,j-1}$$

To pinpoint the position of last reference, we must find the newest counter that does not increase. This can be done by comparing the intra-counter change of adjacent counters. This difference is called the *inter-counter* change, and it is defined to be

$$\Delta y_{ij} \;=\; \begin{cases} \Delta x_{i+1,j} - \Delta x_{i,j} & \text{if } i < j \\ 0 & \text{if } i = j \end{cases}$$

Let us illustrate these definitions with an example. Restricting our focus to the first four elements of the example trace from Section 3.1, the matrices Δx and Δy are

$$\Delta x = \begin{array}{cccc} \{\; a, & b, & c, & \mathbf{a} \;\} \\ \hline 1 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 1 & 1 \\ & & & 1 \end{array} \qquad \Delta y = \begin{array}{cccc} \{\; a, & b, & c, & \mathbf{a} \;\} \\ \hline 0 & 0 & 0 & \mathbf{1} \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{array}$$

Every column of Δy either contains only zeros, or contains a single 1. The former case occurs when the element requested in this column has never been requested before. In the latter case, if the single 1 appears in row i, then the last request for that element was at time i. For example, because $\Delta y_{14} = 1$, the last request for element a before time 4 was at time 1.

Determining the stack distance is now simple, as before. While processing column j of the stream, we infer that the last request for the element e_j occurred at time i by observing that $\Delta y_{ij} = 1$. The stack distance for the jth request is the number of distinct elements that were requested between time i and time j, which is C_{ij}. Recall that the MRC at cache size x is the fraction of requests with stack distance exceeding x. Therefore given all the stack distances, we can easily compute the MRC.

4 Practical Counter Stacks

The idealized counter stack stream defined in Section 3 stores the entire matrix C, so it requires space that is quadratic in the length of the trace. This is actually *more* expensive than storing the original trace. In this section we introduce several ideas that allow us to dramatically reduce the space of counter stacks and streams.

Section 4.1 discusses the natural idea of decreasing the time resolution, i.e., keeping only every d^{th} row and column of the matrix C. Section 4.2 discusses the idea of pruning: eventually a counter may have observed the same set of elements as its adjacent counter, at which point maintaining both of them becomes unnecessary. Finally, Section 4.3 introduces the crucial idea of using probabilistic counters to efficiently and compactly estimate the number of distinct elements seen in the trace.

4.1 Downsampling

The simplest way to improve the space used by counter stacks and streams is to decrease the time resolution. This idea is not novel, and similar techniques have been used in previous work [16].

In our context, decreasing the time resolution amounts to keeping only a small submatrix of C that provides enough data, and of sufficient accuracy, to be useful for applications. For example, one could start a new counter only at every d^{th} position in the trace; this amounts to keeping only every d^{th} row of the matrix C. Next, one could update the counters only at every d^{th} position in the trace; this amounts to keeping only every d^{th} column of the matrix C. We call this process *downsampling*.

Adjacent entries in the original matrix C can differ only by 1, so adjacent entries in the downsampled matrix can differ only by d. Thus, any entry that is missing from the downsampled matrix can be estimated using nearby entries that are present, up to additive error d. For large-scale workloads with billions of distinct elements, even choosing a very large value of d has negligible impact on the estimated stack distances and MRCs.

Our implementation uses a slightly more elaborate form of downsampling because we wish to combine traces that may have activity bursts in disjoint time intervals and

avoid writing columns during idle periods. As well as starting a new counter and updating the old counters after every d^{th} request, we also start a new counter and update the old counters every s seconds with one exception: we do not output a column if the previous s seconds contain no activity. Our experiments reported in Section 7 pick $d = 10^6$ and $s \in \{60, 3600\}$.

4.2 Pruning

Recall that every row of the matrix contains a sequence of values reported by some counter. For any two adjacent counters, the older one (the upper row) will always emit values larger than or equal to the younger one (the lower row). Let us consider the difference of these counters. Initially, at the time the younger one is created, their difference is simply the number of distinct elements seen by the older counter so far. If any of these elements reappears in the trace, the older counter will not increase (as it has seen this element before), but the younger counter will increase, so the difference of the counters shrinks.

If at some point the younger counter has seen every element seen by the older counter, then their difference becomes zero and will remain zero forever. In this case, the younger counter provides no additional information, so it can be deleted. An extension of this idea is that, when the difference between the counters becomes sufficiently small, the younger counter provides negligible additional information. In this case, the younger counter can again be deleted, and its value can be approximated by referring to the older counter. We call this process *pruning*.

The simplest pruning strategy is to delete the younger counter whenever its value differs from its older neighbor by at most p. This strategy ensures that the number of active counters at any point in time is at most M/p. (Recall that M is the number of distinct blocks in the entire trace.) In our current implementation, in order to fix a set of parameters that work well across many workloads of varying sizes, we instead delete the younger counter whenever its value is at least $(1 - \delta)$ times the older counter's value. This ensures that the number of active counters is at most $O(\log(M)/\delta)$. Our experiments reported in Section 7 pick $\delta \in \{0.1, 0.02\}$.

4.3 Probabilistic Counters

Counter stack streams contain the number of distinct blocks seen in the trace between any two points in time (neglecting the effects of downsampling and pruning). The on-disk stream only needs to store this matrix of counts, as the examples in Section 3 suggested. The in-memory counter stack has a more difficult job – it must be able to update these counts while processing the trace, so each counter must keep an internal representation of the set of blocks it has seen.

The naïve approach is for each counter to represent this set explicitly, but this would require quadratic memory usage (again, neglecting downsampling and pruning). A slight improvement can be obtained through the use of Bloom filters [6], but for an acceptable error tolerance, the space would still be prohibitively large. Our approach is to use a tool, called a *probabilistic counter* or *cardinality estimator*, that was developed over the past thirty years in the streaming algorithms and database communities.

Probabilistic counters consume extremely little space and have guaranteed accuracy. The most practical of these is the HyperLogLog counter [18], which we use in our implementation. Each count appearing in our on-disk stream is not the true count of distinct blocks, but rather an estimate produced by a HyperLogLog counter which is correct up to multiplicative factor $1+\epsilon$. The memory usage of each HyperLogLog counter is roughly *logarithmic* in M, with more accurate counters requiring more space. More concretely, our evaluation discussed in Section 7 uses as little as 53 MB of memory to process traces containing over a hundred million requests and distinct blocks.

4.4 LRU Stack Distances

The technique in Section 3.2 for computing stack distances and MRCs using idealized counter stacks can be adapted to use practical counter stacks. The matrices Δx and Δy are defined as before, but are now based on the downsampled, pruned matrix containing probabilistic counts. Previously we asserted that every column of Δy is either all zeros or contains a single 1. This is no longer true. The entry Δy_{ij} now reports the *number* of requests since the counters were last updated whose stack distance was approximately C_{ij}.

To approximate the stack distances of all requests, we process all columns of the stream. As there may be many non-zero entries in the jth column of Δy, we record Δy_{ij} occurrences of stack distance C_{ij} for every i. As before, given all stack distances we can compute the MRC.

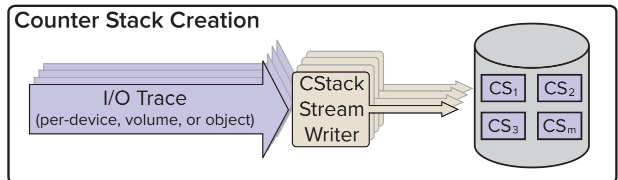
An online version of this approach which does not emit streams can produce an MRC of guaranteed accuracy using provably sublinear memory. In a companion paper [15] we prove the following theorem. The key point is that the space depends polynomially on ℓ and ϵ, the parameters controlling the precision of the MRC, but only logarithmically on N, the length of the trace.

Theorem 1. *The online algorithm produces an estimated MRC that is correct to within additive error ϵ at cache sizes $\frac{1}{\ell}M, \frac{2}{\ell}M, \frac{3}{\ell}M, \ldots, M$ using only*

$O(\ell^2 \log(M) \log^2(N)/\epsilon^2)$ *bits of space, with high probability.*

5 The Counter Stack API

The previous two sections have given an abstract view of counter stacks. In this section we describe the system that we have implemented based on those ideas. The system is a flexible, memory-efficient library that can be used to process traces, produce counter stack streams, and perform queries on those streams. The workflow of applications that use this library is illustrated in Figure 1.



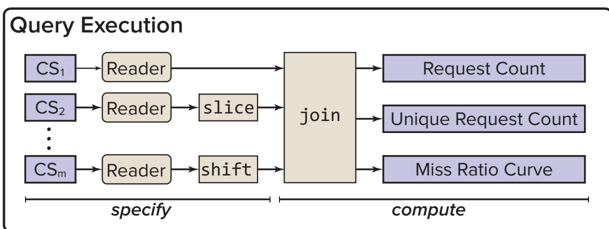


Figure 1: The counter stack library architecture.

5.1 On-disk Streams

The on-disk streams output by the library are produced by periodically outputting a new column of the matrix. As discussed in Section 4, a new column is produced if either d requests have been observed in the trace or s seconds have elapsed (in the trace's time) since the last column was produced, except for idle periods, which are elided. Each column is written to disk in a sparse format to incorporate the fact that pruning may cause numerous entries to be missing.

In addition, the on-disk matrix C includes an extra row, called row R, which records the raw number of requests observed in the stream. That is, C_{Rj} contains the total number of requests processed at the time that the jth column is output. Finally, the on-disk stream also records the trace's time of the current request.

5.2 Compute Queries

The counter stack library supports three computational queries on streams: *Request Count*, *Unique Request Count* and *MRC*.

The first two query operations are straightforward but useful, as we will show in Section 8.4. The Request Count query simply asks for the total number of requests that occur in the stream, which is C_{Rj} where j is the index of the last column. The Unique Request Count query is similar except that it asks for the total number of unique requests, which is C_{1j}.

The most complicated stream operation is the MRC query, which asks for the miss ratio curve of the given stream. This query is processed using the method described in Section 4.4.

5.3 Time Slicing and Shifting

It is often useful to analyze only a subset of a given trace within a specific time interval. We refer to this time-based selection as *slicing*. It is similarly useful when joining traces to alter the time signature by a constant time interval. We refer to this alteration as *shifting*.

The counter stack library supports slicing and shifting as specification operations. Given a stream containing a matrix C, the stream for the time slice between time step i and j is the submatrix with corners at C_{ii} and C_{jj}. Likewise, to obtain the stream for the trace shifted forward/backward s time units, we simply add/subtract s to each of the time indices associated with the rows and columns of the matrix.

5.4 Joining

Given two or more workloads, it is often useful to understand the behavior that would result if they were combined into a single workload. For example, if each workload is an I/O trace of a different process, one may want to investigate the cache performance of those processes with a shared LRU cache.

Counter stacks enable such analyses through the *join* operation. Given two counter stack streams, the desired output of the join operation is what one would obtain by merging the original two traces according to the traces' times, then producing a new counter stack stream from that merged trace. Our library can produce this new stream using only the two given streams, without examining the original traces. The only assumption we require is that the two streams must access disjoint sets of blocks.

The join process would be simple if, for every i, the time of the ith request were the same in both traces; in this case, we could simply add the matrices stored in the two streams. Unfortunately that assumption is implausible, so more effort is required. The main ideas are to:

- *Expand* the two matrices so that each has a row and column for every time that appears in either trace.

time	1:00	1:02	1:05	1:14	1:17
$\mathbf{A}$	a		b		b
$C_{\mathbf{A}}$	1	**1**	2	**2**	2
		0	**1**	**1**	**1**
			1	**1**	1
				0	**1**
					1
$\mathbf{B}$		d		d	
$C_{\mathbf{B}}$	**0**	**1**	**1**	**1**	**1**
		1	**1**	1	**1**
			0	**1**	**1**
				1	**1**
					0
merge	a	d	b	d	b
$C_{\mathbf{A}} + C_{\mathbf{B}}$	1	2	3	3	3
		1	2	2	2
			1	2	2
				1	2
					1

Figure 2: An example illustrating the join operation.

- *Interpolate* to fill in the new matrix entries.
- *Add* the resulting matrices together.

Let us illustrate this process with an example. Consider a trace $\mathbf{A}$ that requests blocks (a, b, b) at times 1:00, 1:05, 1:17, and a trace $\mathbf{B}$ requests blocks (d, d) at times 1:02 and 1:14. The merge of the two traces is as follows:

time	1:00	1:02	1:05	1:14	1:17
$\mathbf{A}$	a		b		b
$\mathbf{B}$		d		d	
merge	a	d	b	d	b

To join these streams, we must expand the matrices in the two streams so that each has five rows and columns, corresponding to the five times that appear in the traces. After this expansion, each matrix is missing entries corresponding to times that were missing in its trace. We fill in those missing entries by an interpolation process: a missing row is filled by copying the nearest row beneath it, and a missing column is filled by copying the nearest column to the left of it. Figure 2 shows the resulting matrices; interpolated values are shown in bold blue.

Pruned counters can sometimes create negative values in Δx. For example, after pruning a counter in row j at time t, the interpolated value of the pruned counter at $t+1$ is set to the nearest row beneath it, representing a younger counter. Often, this lower counter has a smaller value than the pruned counter. The interpolated value at $t+1$ will then be less than its previous value at t, producing a negative intra-counter change. We can avoid introducing negative values in Δx by replacing any negative values in Δx by the nearest nonnegative value beneath it. This replacement has the same effect of changing the value of the pruned counter to the lower counter in column t prior to calculating the intra-counter change for the column representing $t+1$.

6 Error and Uncertainty

While each of the optimizations described in Section 4 dramatically reduce the storage requirements of counter stacks, they may also introduce uncertainty and error into the final calculations. In this section, we discuss potential sources of error, as well as how to modify the different operations described in Section 3 to compute lower and upper bounds on the stack distances.

6.1 Counter Error

HyperLogLog counters introduce error in two ways: count estimation and simultaneous register updates. HyperLogLog counters report a count of distinct elements that is only correct up to multiplicative factor ϵ, which is determined by a precision parameter. This uncertainty produces deviation from the true MRC and can be controlled by increasing the precision of the HyperLogLog counters, at the cost of a greater memory requirement.

Simultaneous register updates introduce a subtler form of error. A HyperLogLog counter estimates unique counts by taking the harmonic mean of a set of internal variables called *registers*. Due to the design of HLLs, sometimes a register update might cause the older counter to increase in value more than the younger counter. This phonemoneon leads to negative updates in Δy, because older counters are expected to change more slowly than younger counters. Theorem 1 implies that the negative entries in the Δy matrix introduced by simultaneous register updates are offset by corresponding over-estimates when register modifications between counters are not simultaneous.

In some cases, the histogram of stack distances may accumulate enough negative entries that there are bins with negative counts. The cumulative sum of such a histogram will result in a non-monotonic MRC. We can enforce a monotonic MRC by accumulating any negative histogram bins in a separate counter, carrying the difference forward in the cumulative sum and discounting positive bins by the negative count. In practice, negative histogram entries make up less then one percent of the reported stack distances, with little to no visible effect on the accumulated MRC.

6.2 Downsampling Uncertainty

Whereas the scheme of Section 3.2 computes stack distances exactly, the modified scheme of Section 4.4 only computes approximations. This uncertainty in the stack distances is caused by downsampling, pruning and use of probabilistic counters. To illustrate this, consider the example shown in Figure 3, and for simplicity let us ignore pruning and any probabilistic error.

At every time step j, the finite differencing scheme uses the matrix Δy to help estimate the stack distances for all requests that occurred since time step $j-1$. More concretely, if such a request increases the $(i+1)^{\text{th}}$ counter but does not increase the ith counter, then we know that the most recent occurrence of the requested block lies somewhere between time step i and time step $i+1$. Since there may have been many requests between time i and time $i+1$, we do not have enough information to determine the stack distance exactly, but we estimate it up to additive error d (the downsampling factor). A careful analysis can show that the request must have stack distance at least $C_{i+1,j-1}+1$ and at most C_{ij}.

7 Evaluation

In this section we empirically validate two claims: (1) the time and space requirements of counter stack processing are sufficiently low that it can be used for online analysis of real storage workloads, and (2) the technique produces accurate, meaningful results.

We use a well-studied collection of storage traces released by Microsoft Research in Cambridge (MSR) [26] for much of our evaluation. The MSR traces record the disk activity (captured beneath the file system cache) of 13 servers with a combined total of 36 volumes. Notable workloads include a web proxy (`prxy`), a filer serving project directories (`proj`), a pair of source control servers (`src1` and `src2`), and a web server (`web`). The raw traces comprise 417 million records and consume just over 5 GB in compressed CSV format.

We compare our technique to the 'ground truth' obtained from full trace analysis (using *trace trees*, the tree-based optimization of Mattson's algorithm [23, 27]), and, where applicable, to a recent approximation technique [37] which derives estimated MRCs from *average footprints* (see Section 9 for more details). For fairness, we modify the original implementation [13] by using a sparse dictionary to reduce memory overhead.

7.1 Performance

The following experiments were conducted on a Dell PowerEdge R720 with two six-core Intel Xeon processors and 96 GB of RAM. Traces were read from high-performance flash to eliminate disk IO bottlenecks.

Fidelity	Time	Memory	Throughput	Storage
low	17.10 m	78.5 MB	2.31M reqs/sec	747 KB
high	17.24 m	80.6 MB	2.29M reqs/sec	11 MB

Table 1: The resources required to create low and high fidelity counter stacks for the combined MSR workload (64 MB heap).

Throughout this section we present figures for both 'low' and 'high' fidelity streams. We control the fidelity by adjusting the number of counters maintained in each stream; the parameters used in these experiments represent just two points of a wide spectrum, and were chosen in part to illustrate how accuracy can be traded for performance to meet individual needs.

We first report the resources required to convert a raw storage trace to a counter stack stream. The memory footprint for the conversion process is quite modest: converting the entire set of MSR traces to high-fidelity counter stacks can be done with about 80 MB of RAM [1]. The processing time is low as well: our Java implementation can convert a trace to a high-fidelity stream at a rate of 2.3 million requests per second with a 64 MB heap and 2.7 million requests per second with a 256 MB heap.

The size of counter stack streams can also be controlled by adjusting fidelity. Ignoring write requests, the full MSR workload consumes 2.9 GB in a compressed, binary format. We can reduce this to 854 MB by discarding latency values and capping timestamp resolutions at one second, and we can shave off another 50 MB through domain-specific compaction techniques like delta-encoding time and offset values. But as Table 1 shows, this is more than 70 times larger than a high-fidelity counter stack representation.

The compression achieved by counter stack streams is workload-dependent. High-fidelity streams of the MSR workloads are anywhere from 12 (`hm`) to 1,024 (`prxy`) times smaller than their compressed binary counterparts, with larger traces tending to compress better. A stream of the combined traces consumes just over 1.5 MB per day, meaning that weeks or even months of workload history can be retained at very reasonable storage costs.

Once a trace has been converted to a counter stack stream, performing queries is very quick. For example, an MRC for the entire week-long MSR trace can be com-

[1] This is not a lower bound. Additional reductions can be achieved at the expense of increased garbage collection activity in the JVM; for example, enforcing a heap limit of 32 MB increases processing time for the high-fidelity counter stack by about 30% and results in a peak resident set size of 53 MB.

C			
	10	20	50
		15	50
			40
$\mathbf{R}$	100	200	300

$\rightarrow$

Δx			
	10	10	30
		15	35
			40
$\mathbf{\Delta R}$	100	100	100

$\rightarrow$

Δy			
	$90\ (1, 10)$	$5\ (1, 20)$	$5\ (16, 50)$
		$85\ (1, 15)$	$5\ (1, 50)$
			$60\ (1, 40)$

Figure 3: An example of computing stack distances using a downsampled matrix. The entries of Δy show the number of requests and the parenthesized values show the bounds on the stack distances that we can infer for those requests.

puted from the counter stack stream in just seconds, with negligible memory overheads. By comparison, computing the same MRC using a trace tree takes about an hour and reaches a peak memory consumption of 92 GB, while the average footprint technique requires 8 and a half minutes and 23 GB of RAM.

7.2 Accuracy

Figure 4 shows miss ratio curves for each of the individual workloads contained in the MSR traces as well as the combined `master` trace; superimposed on the baseline curves (showing the exact MRCs) are the curves computed using footprint averages and counter stacks. Some of the workloads feature MRCs that are notably different from the convex functions assumed in the past [35]. The `web` workload is the most obvious example of this, and it is also the workload which causes the most trouble for the average footprint technique.

Figure 5 shows three examples of MRCs produced by *joining* individual counter stacks. The choice of workloads is somewhat arbitrary; we elected to join workloads of commensurate size so that each would contribute equally to the resulting merged MRC. As described in Section 5.4, the join operation can introduce additional uncertainty due to the need to infer the values of missing counters, but the effects are not prominent with the high-fidelity counter stacks used in these examples.

We performed an analysis of curve errors at different fidelities, with `verylow` ($\delta = 0.46$, $d = 19M$, $s = 32K$) at one extreme and `high` ($\delta = 0.01$, $d = 1M$, $s = 60$) at the other. To measure curve error, we use the Mean Absolute Error (MAE) between a given curve and its ground-truth counterpart. The MAE is defined as the average absolute difference between two series mrc and mrc', or $\frac{1}{N}\sum|mrc(x)-mrc'(x)|$. Because MRCs range between 0 and 1, the MAEs are also confined to the same range, where a value of 0 implies perfectly corresponding curves. At the other extreme, it is difficult to know what constitutes a "bad" MAE because it is unlikely to be close to 1 except in singular cases. For example, the MAE between the `hm` and the `ts` Mattson curves is only 0.15. For the high fidelity counter stacks, we observe MAEs between 0.002 and 0.02, and for the average footprint algorithm, we observe MAEs between 0.001 and 0.04.

We find that curve error under compression is highly workload-dependent. We observed the largest errors on "jagged" workloads with sharp discontinuities, such as `src1` and `web`, while workloads with "flatter" MRCs such as `stg` and `usr` are almost invariant to compression. Figure 6 summarizes our findings on two such workloads. On the left, we illustrate the difference in the change in error as fidelity decreases for a jagged workload, `src1`, and a flat workload, `usr`. On the right, we show the smoothing effect of decreasing the counter stack fidelity by comparing the `verylow` and `high` fidelity curves against Mattson on `src1`.

8 Workload Analysis

We have shown that counter stacks can be used to produce accurate MRC estimations in a fraction of the time and space used by existing techniques. We now demonstrate some of the capabilities of the counter stack query interface through a series of case studies of the MSR traces.

8.1 Combined Workloads

Hit rates are often used to gauge the health of a storage system: high hit rates are considered a sign that a system is functioning properly, while poor hit rates suggest that tuning or configuration changes may be required. One problem with this simplistic view is that the combined hit rates of multiple independent workloads can be dominated by a single workload, thereby hiding potential problems.

We find this is indeed the case for the MSR traces. The `prxy` workload features a small working set and a high activity rate – it accesses only 2 GB of unique data over the entire week but issues 15% of all read requests in the combined trace. Table 2 puts this in perspective: the combined workload achieves a hit rate of 50% with a 550 GB cache; more than 250 GB of additional cache capacity would be required to achieve this same hit rate without the `prxy` workload. This illustrates why *combined hit rate is not an adequate metric of system behavior.* Diagnostic tools which present hit rates as an indicator of storage well-being should be careful to consider workloads independently as well as in combination.

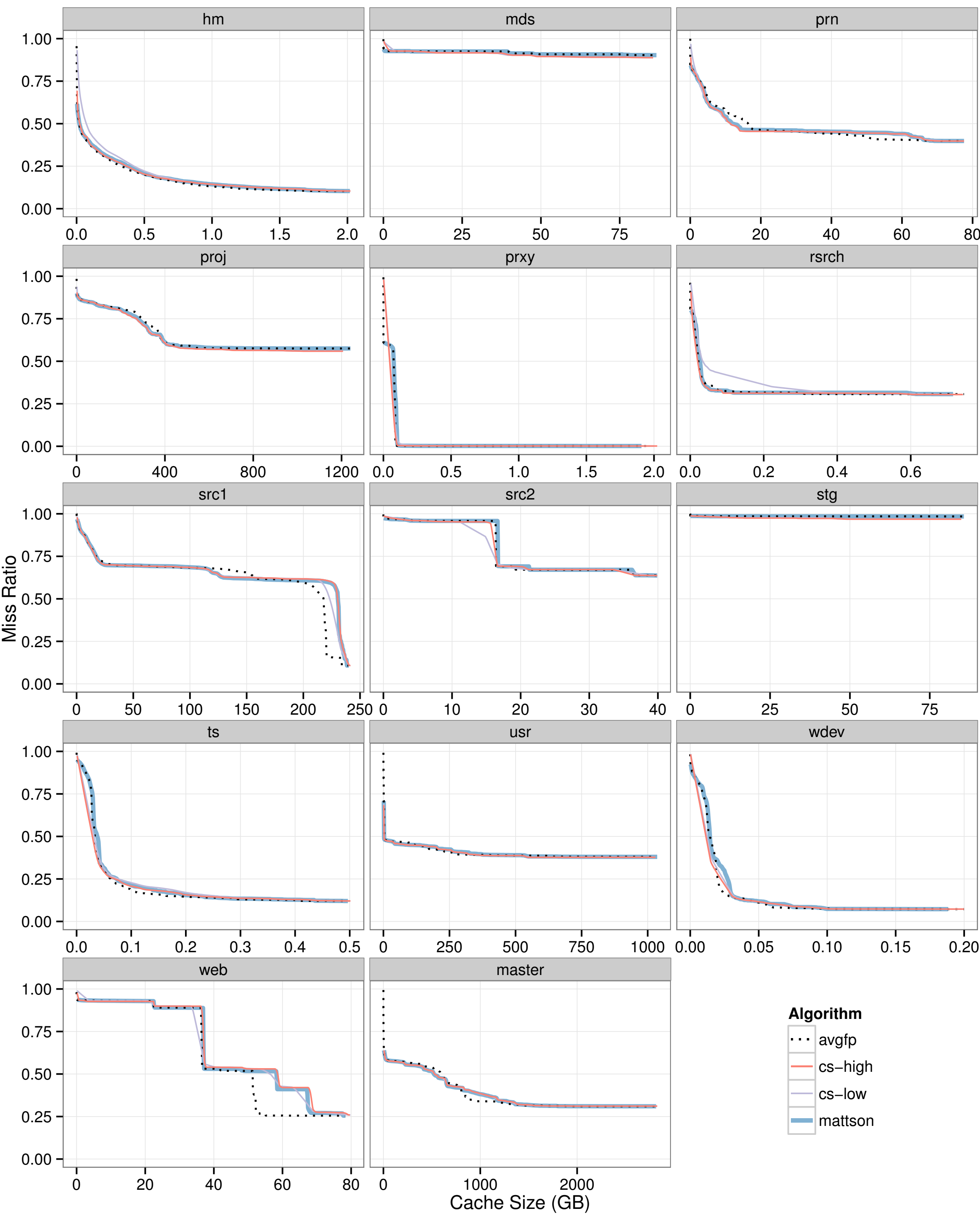


Figure 4: MSR miss ratio curves.

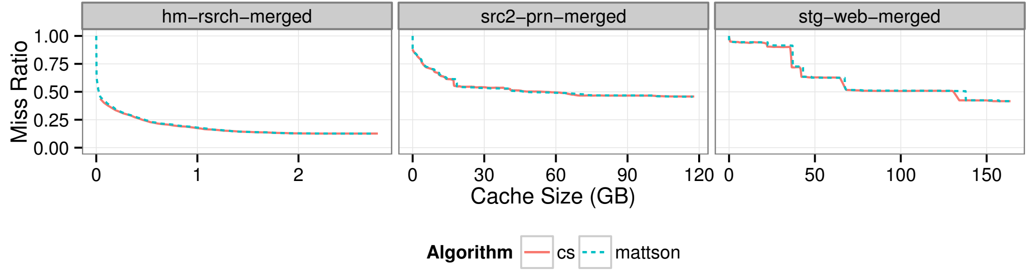


Figure 5: MRCs for various combinations of MSR workloads (produced by the *join* operation).

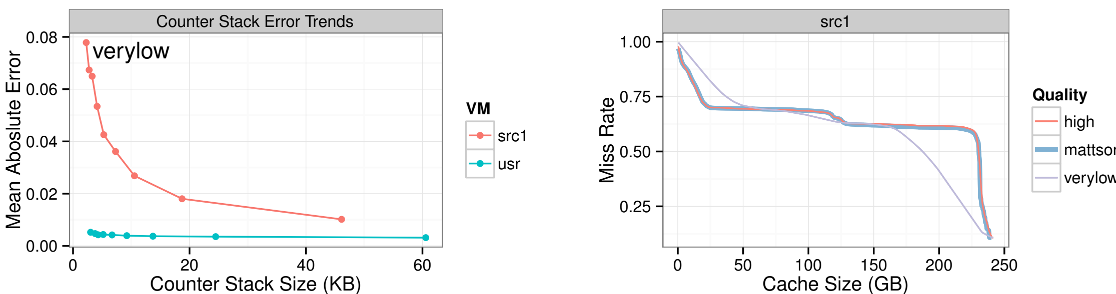


Figure 6: The qualitative effect of counter stack fidelity is workload-dependent. On the left, we show the curve error and file sizes of different fidelities. The `usr` workload is robust to compression to very low fidelity, while the `src1` workload degrades progressively. On the right, we show the visual outcome of compression to both *high* and *verylow* fidelity on `src1`.

Desired Hit Rate	Required Cache Size	
	With `prxy`	Without `prxy`
30%	2.5 GB	21.6 GB
40%	19.2 GB	525.5 GB
50%	566.6 GB	816.0 GB

Table 2: Cache sizes required to obtain desired hit rates for combined MSR workloads with and without `prxy`.

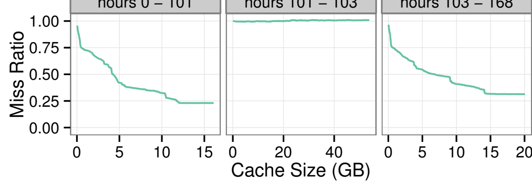


Figure 7: Time-sliced `prn` workload.

8.2 Erratic Workloads

MRCs can be very sensitive to anomalous events. A one-off bulk read in the middle of an otherwise cache-friendly workload can produce an MRC with high miss rates, arguably mischaracterizing the workload. We wrote a simple script that identifies erratic workloads by searching for hour-long slices with unusually high miss ratios. The script found several workloads, including `mds`, `stg`, `ts`, and `prn`, whose week-long MRCs are dominated by just a few hours of intense activity.

Figure 7 shows the effect these bursts can have on workload performance. The full-week MRC for `prn` (Figure 4) shows a maximum achievable hit rate of 60% at a cache size of 83 GB. The workload features a two-hour read burst starting 102 hours into the trace which accounts for 29% of the total requests and 69% of the unique blocks. Time-sliced MRCs before and after this burst feature hit rates of 60% at cache sizes of 10 GB and 12 GB, respectively. This is a clear example of how *anomalous events can significantly distort MRCs*, and it shows why it is important to consider MRCs over various intervals in time, especially for long-lived workloads.

8.3 Conflicting Workloads

Many real-world workloads exhibit pronounced diurnal patterns: interactive workloads typically reflect natu-

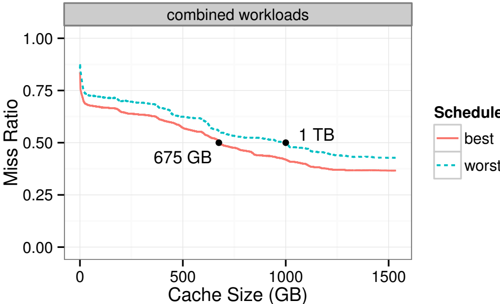


Figure 8: Best and worst time-shifted MRCs for MSR workloads (excluding prxy). We omit cache sizes greater than 1.5 TB to preserve details in the plot.

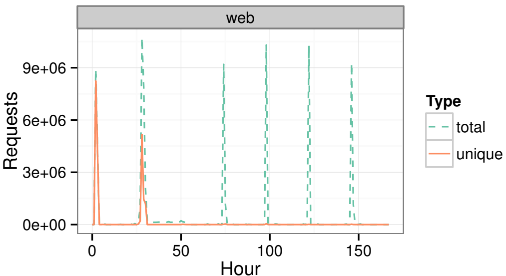


Figure 9: web total and unique requests per hour.

ral trends in business hours, while automatic workloads are often scheduled at regular intervals throughout the day [17, 22, 29]. When such workloads are served by the same shared storage, it makes sense to try to limit the degree to which they interfere with one another.

The time-shifting functionality of counter stacks provides a powerful tool for exploring coarse-grain scheduling of workloads. To demonstrate this, we wrote a script which computes the MRCs of the combined MSR trace (excluding prxy) in which the start times of a few of the larger workloads (proj, src1, and usr) are shifted by up to six hours. Figure 8 plots the best and worst MRCs computed by this script. As is evident, *workload scheduling can significantly affect hit rates*. In this case, shifting workloads by just a few hours changes the capacity needed for a 50% hit rate by almost 50%.

8.4 Periodic Workloads

MRCs are good at characterizing the raw capacity needed to accommodate a given working set, but they provide very little information about how that capacity is used over time. In environments where many workloads share a common cache, this lack of temporal information can be problematic. For example, as Figure 4 shows, the entire working set of web is less than 80 GB, and it exhibits a hit rate of 75% with a dedicated cache at this size. However, as shown in Figure 9, the workload is highly periodic and is idle for all but a few hours every day.

This behavior is characteristic of automated tasks like nightly backups and indexing jobs, and it can be problematic because *periodic workloads with long reuse distances tend to perform poorly in shared caches*. The cost of this is twofold: first, the periodic workloads exhibit low hit rates because their long reuse distances give them low priority in LRU caches; and second, they can penalize other workloads by repeatedly displacing 'hotter' data. This is exactly what happens to web in a cache shared with the rest of the MSR workloads: despite its modest working set size and high locality, it achieves a hit rate of just 7.5% in a 250 GB cache and 20% in a 500 GB cache.

Scan-resistant replacement policies like ARC [24] and CAR [3] offer one defense against this poor behavior by limiting the cache churn induced by periodic workloads. But a better approach might be to the exploit the highly regular nature of such workloads – assuming they can be identified – through intelligent prefetching. Counter stacks are well-suited for this task because they make it easy to detect periodic accesses to non-unique data. While this alone would not be sufficient to implement intelligent prefetching (because the counters do not indicate *which* blocks should be prefetched), it could be used to alert the system of the recurring pattern and initiate the capture of a more detailed trace for subsequent analysis.

8.5 Zipfian Workloads

We end with a brief discussion of synthetic workload generators like FIO [2] and IOMeter [32]. These tools are commonly used to test and validate storage systems. They are capable of generating IO workloads based on parameters describing, among other things, read/write mix, queue depth, request size, and sequentiality. The simpler among them support various combinations of random and sequential patterns; FIO recently added support for pareto and zipfian distributions, with the goal of better approximating real-world workloads.

Moving from uniform to zipfian distributions is a step in the right direction. Indeed, many of the MSR workloads, including hm, mds, and prn, exhibit roughly zipfian distributions. However, as is evident in Figure 4, the MRCs of these workloads vary dramatically. Figure 10 plots the MRC of a perfectly zipfian workload produced by FIO alongside two permutations of the same workload; as expected, request ordering has a significant impact on locality and cache behavior. These figures show that *syn-*

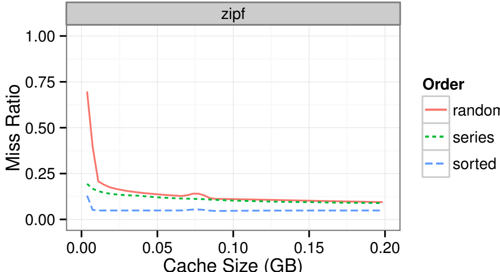


Figure 10: MRCs for three permutations of a single zipfian distribution: `random`, `series` (a concatenation of sorted series of unique requests), and `sorted` (truncated to preserve detail).

thetic zipfian workloads do not necessarily produce 'realistic' MRCs, emphasizing the importance of using real-world workloads when evaluating storage performance.

9 Related Work

Mattson et al. [23] defined stack distances and presented a simple $O(NM)$ time, $O(M)$ space algorithm to calculate them. Bennett and Kruskal [4] used a tree-based implementation to bring the runtime to $O(N \log(N))$. Almási et al. improved this to $O(N \log(M))$, and Niu et al. [27] introduced a parallel algorithm.

A different line of work explores techniques to efficiently approximate stack distances. Eklov and Hagersten [16] proposed a method to estimate stack distances based on sampling. Ding and Zhong [14] use an approximation technique inspired by the tree-based algorithms. Xiang et al. [37] define the footprint of a given trace window to be the number of distinct blocks occurring in the window. Using reuse distances, they estimate the average footprint across a logarithmic scale of window lengths. Xiang et al. [38] then develop a theory connecting the average footprint and the miss ratio, contingent on a regularity condition they call the *reuse-window hypothesis*. In comparison, counter stacks use dramatically less memory while producing MRCs with comparable accuracy.

A large body of work from the storage community explores methods for representing workloads concisely. Chen et al. [9] use machine learning techniques to extract workload features, Tarasov et al. [36] describe workloads with feature matrices, and Delimitrou et al. [11] model workloads with Markov Chains. These representations are largely incomparable to counter stacks – they capture many details that are not preserved in counter stack streams, but they discard much of the temporal information required to compute accurate MRCs.

Many domain-specific compression techniques have been proposed to reduce the cost of storing and processing workload traces. These date back to Smith's stack deletion [33] and include Burtscher's VPC compression algorithms [8]. They generally preserve more information than counter stacks but achieve lower compression ratios. They do not offer new techniques for MRC computation.

10 Conclusion

Sizing the tiers of a hierarchical memory system and managing data placment across them is a difficult, workload dependent problem. Techniques such as miss ratio curve estimation have existed for decades as a method of modeling workload behaviors offline, but their computational and memory overheads have prevented their incorporation as a means to make live decisions in real systems. Even as an offline tool, practical issues such as the overheads associated with trace collection and storage often prevent the sharing and analysis of memory access traces.

Counter stacks provide a powerful software tool to address these issues. They are a compact form of locality characterization that allow workloads to be studied in new interactive ways, for instance by searching for anomalies or shifting workloads to identify pathological load possibilities. They can also be incorporated directly into system design as a means of making more informed and workload-specific decisions about resource allocation across multiple tenants.

While the design and implementation of counter stacks described in this paper have been motivated through the design of an enterprise storage system, the techniques are relevant in other domains, such as processor architecture, where the analysis of working set size over time and across workloads is critical to the design of efficient, high-performance systems.

References

[1] G. S. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on memory system performance (MSP '02)*, pages 37–43, 2002.

[2] J. Axboe. Fio–flexible I/O tester, 2011.

[3] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

[4] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

[5] M. Blaze. NFS tracing by passive network monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, 1992.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] A. D. Brunelle. Block I/O Layer Tracing: blktrace. *HP, Gelato-Cupertino, CA, USA*, 2006.

[8] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The vpc trace-compression algorithms. *Computers, IEEE Transactions on*, 54(11):1329–1344, 2005.

[9] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 43–56. ACM, 2011.

[10] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: scalable high-performance storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*, pages 17–31. USENIX Association, 2014.

[11] C. Delimitrou, S. Sankar, K. Vaid, and C. Kozyrakis. Decoupling datacenter studies from access to large-scale applications: A modeling approach for storage workloads. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 51–60. IEEE, 2011.

[12] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[13] C. Ding. Program locality analysis tool. `https://github.com/dcompiler/loca`, 2014.

[14] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, pages 245–257. ACM, 2003.

[15] Z. Drudi, N. J. A. Harvey, S. Ingram, A. Warfield, and J. Wires. Approximating MRCs using counter stacks. Technical report, Coho Data, 2014.

[16] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 55–65. IEEE, 2010.

[17] D. Ellard, J. Ledlie, P. Malkani, and M. I. Seltzer. Passive NFS tracing of email and research workloads. In *FAST*. USENIX, 2003.

[18] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, 0(1), 2008.

[19] B. Jacob, P. Larson, B. Leitao, and S. da Silva. SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems. *IBM Redbook*, 2008.

[20] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *ASPLOS*, pages 14–24. ACM, 2006.

[21] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122. IEEE Computer Society, 2004.

[22] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX Annual Technical Conference*, pages 213–226, 2008.

[23] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[24] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[25] P. Mochel. The sysfs Filesystem. In *Linux Symposium*, page 313, 2005.

[26] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[27] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1284–1294. IEEE, 2012.

[28] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance,

runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[29] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: Recapitulating storage for virtual desktops. In *FAST*, pages 31–45, 2011.

[30] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *POPL*, pages 55–61. ACM, 2007.

[31] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, pages 165–176. ACM, 2004.

[32] J. Sievert. Iometer: The I/O performance analysis tool for servers, 2004.

[33] A. J. Smith. Two methods for the efficient analysis of memory address trace data. *Software Engineering, IEEE Transactions on*, 3(1):94–101, 1977.

[34] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic resource allocation for database servers running on virtual storage. In *FAST*. USENIX, 2009.

[35] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *Computers, IEEE Transactions on*, 41(9):1054–1068, 1992.

[36] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. *FAST*, 2012.

[37] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 350–360. IEEE, 2011.

[38] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 343–356. ACM, 2013.

[39] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *OSDI*, pages 103–116. ACM, 2006.

[40] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Low cost working set size tracking. In *Annual Technical Conference*, pages 223–228. USENIX, 2011.

[41] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI*, pages 255–266. ACM, 2004.

[42] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, pages 177–188. ACM, 2004.

[43] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.

Pelican: A building block for exascale cold data storage

Shobana Balakrishnan, Microsoft Research
Richard Black, Microsoft Research
Austin Donnelly, Microsoft Research
Paul England, Microsoft Research
Adam Glass, Microsoft Research
Dave Harper, Microsoft Research
Sergey Legtchenko, Microsoft Research
Aaron Ogus, Microsoft
Eric Peterson, Microsoft Research
Antony Rowstron, Microsoft Research

Abstract

A significant fraction of data stored in cloud storage is rarely accessed. This data is referred to as *cold data*; cost-effective storage for cold data has become a challenge for cloud providers. Pelican is a rack-scale hard-disk based storage unit designed as the basic building block for exabyte scale storage for cold data. In Pelican, server, power, cooling and interconnect bandwidth resources are provisioned by design to support cold data workloads; this *right-provisioning* significantly reduces Pelican's total cost of ownership compared to traditional disk-based storage.

Resource right-provisioning in Pelican means only 8% of the drives can be concurrently spinning. This introduces complex resource management to be handled by the Pelican storage stack. Resource restrictions are expressed as constraints over the hard drives. The data layout and IO scheduling ensures that these constraints are not violated. We evaluate the performance of a prototype Pelican, and compare against a traditional resource over-provisioned storage rack using a cross-validated simulator. We show that compared to this over-provisioned storage rack Pelican performs well for cold workloads, providing high throughput with acceptable latency.

1 Introduction

Cloud storage providers are experiencing an exponential growth in storage demand. A key characteristic of much of the data is that it is rarely read. This data is commonly referred to as being *cold data*. Storing data that is read once a year or less frequently in online hard disk based storage, such as Amazon S3, is expensive, as the hardware is provisioned for low latency access to the data [7]. This has led to a push in industry to understand and build out cloud-scale tiers optimized for storing cold data, for example Amazon Glacier [1] and Facebook Cold Data Storage [8]. These systems attempt to minimize up front capital costs of buying the storage, as well as the costs of running the storage.

In this paper we describe Pelican, a prototype rack-scale storage unit that forms a basic building block for building out exabyte-scale cold storage for the cloud. Pelican is a converged design, with the mechanical, hardware and storage software stack being co-designed. This allows the entire rack's resources to be carefully balanced, with the goal of supporting only a cold data workload. We have designed Pelican to target a peak sustainable read rate of 1 GB per second per PB of storage (1 GB/PB/sec), assuming a Pelican stores 1 GB blobs which are read in their entirety. We believe that this is higher than the actual rate required to support a cold data workload. The current design provides over 5 PB of storage in a single rack, but at the rate of 1 GB/PB/s the entire contents of a Pelican could be transferred out every 13 days.

All aspects of the design of Pelican are *right-provisioned* to the expected workload. Pelican uses a 52U standard-sized rack. It uses two servers connected using dual 10 Gbps Ethernet ports to the data center network, providing an aggregate of 40 Gbps of full-duplex bandwidth. It has 1,152 archive-grade hard disks packed into the rack, and using currently available drives of average size 4.55 TB provides over 5PB of storage. Assuming a goodput of approximately 100 MB/s for a hard disk drive, including redundancy overheads, then only 50 active disks are required to sustain 40 Gbps.

A traditional storage rack would be provisioned for peak performance, with sufficient power and cooling to allow all drives to be concurrently spinning and active. In Pelican there is sufficient cooling to allow only 96 drives to be spun up. All disks which are not spun up are in standby mode, with the drive electronics powered but the platters stationary. Likewise we have sufficient power for only 144 active spinning drives. The PCIe-bus is stretched out across the entire rack, and we provision it to have 64 Gbps of bandwidth at the root of the

PCIe-bus, much less than 1 Gbps per drive. This careful provisioning reduces the hardware required in the rack and increases efficiency of hardware layout within the rack. This increases storage density; the entire rack drive density is significantly higher than any other design we know, and reduces peak and average power consumption. All these factors result in a hardware platform significantly cheaper to build and operate.

This *right-provisioning* of the hardware, yet providing sufficient resources to satisfy the workloads, differentiates us from prior work. Massive Arrays of Idle Disks (MAID) systems [5] and storage systems that achieve power-proportionality [15] assume that there is sufficient power and cooling to have all disks spinning and active when required. Therefore, they simply provide a power saving during operation but still require the hardware and power to be provisioned for the peak. For a Pelican rack the peak power is approximately 3.7 kW, and average power is around 2.6 kW. Hence, a Pelican provides both a lower capital cost per disk as well as a lower running cost, whereas the prior systems provide just a lower running cost.

However, to benefit from our hardware design we need a software storage stack that is able to handle the restrictions of having only 8% of the disks spinning concurrently. This means that Pelican operates in a regime where spin up latency is the modern day equivalent of disk seek latency. We need to design the Pelican storage stack to handle the disk spin up latencies that are on the order of 10 seconds.

The contributions of this paper are that we describe the core algorithms of the Pelican software stack that give Pelican good performance even with hardware restricted resources, in particular how we handle data layout and IO scheduling. These are important to optimize in order to achieve high throughput and low per operation latency. Naive approaches yield very poor performance but carefully designing these algorithms allows us to achieve high-throughput with acceptable latency.

To support data layout and IO scheduling Pelican uses groups of disks that can be considered as a schedulable unit, meaning that all disks in the group can be spun up concurrently without violating any hardware restrictions. Each resource restriction, such as power, cooling, vibration, PCIe-bandwidth and failure domains, is expressed as constraints over sets of physical disks. Disks are then placed into one of 48 groups, ensuring that all the constraints are maintained. Files are stored within a single group and, as a consequence, the IO scheduler needs to schedule in terms of 48 groups rather than 1,152 disks. This allows the stack to handle the complexity of the right-provisioning.

We have built out a prototype Pelican rack, and we present experimental results using that rack. In order to allow us to compare to an over-provisioned storage rack we use a rack-scale simulator to compare the performance. We cross-validate the simulator against the full Pelican rack, and show that it is accurate. The results show that we are able to sustain a good throughput and control latency of access for workloads up to 1 GB/PB/sec.

The rest of this paper is organized as follows. Section 2 provides an overview of the Pelican hardware, and describes the data layout and IO Scheduler. Section 3 describes a number of issues we discovered when using the prototype hardware, including issues around sequencing the power-up with hardware restrictions. In Section 4 we evaluate the performance of Pelican and the design choices. Related work is detailed in Section 5. Finally, Section 6 discusses future work and Section 7 concludes.

2 A Pelican Rack

A Pelican stores unstructured, immutable chunks of data called *blobs* and has a key-value store interface with `write`, `read` and `delete` operations. Blobs in the size range of 200 MB to 1 TB are supported, and each blob is uniquely identified by a 20 byte key supplied with the operation.

Pelican is designed to store blobs which are infrequently accessed. Under normal operation, we assume that for the first few months blobs will be written to a Pelican, until a target capacity utilization is hit, and from then on blobs will be rarely read or deleted during the rest of the lifetime of the Pelican until it is decommissioned. Hence the normal mode of operation for the lifetime of the rack will be servicing reads, and generating internal repair traffic when disks fail or are replaced.

We also assume that Pelican is used as a lower tier in a cloud-based storage system. Data is staged in a higher tier awaiting transfer to a Pelican, and the actual time when the data is migrated is under the control of the target Pelican. This means that writes can always occur during quiescent or low load periods. Therefore, we focus on the performance of a Pelican for read-dominated workloads.

We start by providing an overview of the Pelican hardware, before describing in detail how the storage stack performs data placement and request scheduling to handle the right-provisioned hardware.

2.1 Pelican hardware

Pelican is a 52U rack filled with 1,152 archival class 3.5” SATA disks. Pelican uses a new class of archival drive manufactured for cold storage systems. The disks are placed in trays of 16 disks. The rack contains six 8U chassis each containing 12 trays (192 disks) organized

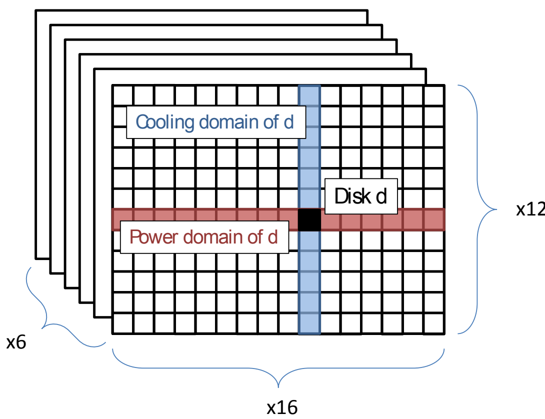


Figure 1: Schematic representation of the Pelican rack

in 2 rows of 6 trays. Each tray is approximately 4U high and the disks are inserted as 8 pairs back-to-back and vertically mounted in the tray. Each chassis has a backplane connecting the 12 horizontally laid trays. Figure 1 shows a schematic representation of a Pelican, a cuboid of 6 (width) x 16 (depth) x 12 (height) disks where the power is shared by disks in the same row and cooling shared by disks in the same column. The backplane supplies power to the 72 trays, and this is currently configured to be sufficient to support two active spinning or spinning up disks. A tray is an independent *power domain*. Due to the nature of power distribution, and the use of electronic fuses, we provision the system to support a symmetrical power draw across all trays. In-rack cooling uses multiple forced air flow channels, each cooling a section of multiple trays, the *cooling domain*. Each air flow is shared by 12 disks. There are 96 independent cooling domains in the rack, and each is currently calibrated to support cooling only one active or spinning up disk. Pelican supports the cooling of 96 drives, but there is sufficient power to have 144 drives spinning. The number of cooling domains and trays is in part driven by convenience for the mechanics and physical layout. Given that approximately 50 drives need to be active in order to allow a Pelican to generate 40Gbps of network traffic, having 96 drives active at any one time, allows us to be servicing requests concurrently with spinning up drives for queued requests.

The SATA disks in the trays are connected to a Host Bus Adapter (HBA) on the tray. The tray HBAs connect to a PCIe switch on the backplane. This supports *virtual switching*, allowing the physical switch to be partitioned into two virtual switches, each with a root port connected to one of the two servers. Each HBA is connected to one of the two virtual switches; this selection is dynamic and under software control.

Each of the six chassis are connected to both servers (12 cables in total) and each PCIe hierarchy provides only 64 Gbps bandwidth at its root. Under normal operation the rack is vertically partitioned into two halves (of 36 trays) with no shared cooling or power constraints. Each server can therefore independently and concurrently handle reads and writes to its half of the rack. However, if a server fails, the PCIe virtual switches can be reconfigured so all disks attach to a single server. Hence, servers, chassis, trays and disks are the *failure domains* of the system.

This right-provisioning of power, cooling and internal bandwidth resources permits more disks per rack and reduces total cost of ownership. The cost reduction is considerable. However, the storage software stack must ensure that the individual resource demands do not exceed the limits set by the hardware.

2.2 Pelican Software Storage Stack

The software storage stack has to minimize the impact on performance of the hardware resource restrictions. Before describing the data layout and IO scheduling algorithms used in Pelican we more formally define the different resource domains.

2.2.1 Resource domains

We assume that each disk uses resources from a set of *resource domains*. Resource domains capture right-provisioning: a domain is only provisioned to supply its resource to a subset of the disks simultaneously. Resource domains operate at the unit of disks, and a single disk will be in multiple resource domains (exactly one for each resource).

Disks that are in the same resource domain for any resource are *domain-conflicting*. Two disks that share no common resource domains are *domain-disjoint*. Disks that are domain-disjoint can move between disk states *independently*, and it is guaranteed there will be no overcommitting of any resources. Disks which are domain-conflicting cannot move independently between states, as doing so could lead to resource over-commitment. Most normal storage systems provisioned for peak performance only take into account only failure domains. With resource right-provisioning we increase considerably the number of domains and so the complexity.

In order to allow the scheduling and the placement to be efficient, we express resource domains as constraints over the disks. We classify the constraints as *hard* or *soft*. Violating a hard constraint causes either short- or long-term failure of the hardware. For example, power and cooling are hard constraints. Violating the power constraint causes an electronic fuse to trip that means drives are not cleanly unmounted, and potentially dam-

aging the hardware. A tray becomes unavailable until the electronic fuse (automatically) resets. Violating the cooling constraint bakes the drives, reducing their lifetime. In contrast, violating a soft constraint does not lead to failure but instead causes performance degradation or inefficient resource usage. For example, bandwidth is a soft constraint: the PCIe bus is a tree topology, and violating the bandwidth constraint simply results in link congestion resulting in a throughput drop.

The Pelican storage stack is designed to enforce operation within the hardware constraints with performance. The Pelican storage stack uses the following constraints: *(i)* one disk spinning or spinning up per cooling domain; *(ii)* two disks spinning or spinning up per power domain; *(iii)* shared links in the PCIe interconnect hierarchy; and *(iv)* disks located back-to-back in a tray share a vibration domain. Like other storage systems we also have failure domains, in our case for disks, trays and chassis.

2.2.2 Data layout

In order to provide resiliency to failures, each blob is stored over a set of disks selected by the data layout algorithm. Blob placement is key as it impacts the concurrency of access to blobs.

When a blob is to be written into a Pelican, it is split into a sequence of 128 kB data fragments. For each k fragments we generate r additional fragments containing redundancy information using a Cauchy Reed-Solomon erasure code [4] and store the $k+r$ fragments. We use a systematic code, which means if the k original fragments are read back then the input can be regenerated by simply concatenating these fragments[1]. Alternatively, a reconstruction read can be performed by reading *any* k fragments, allowing the reconstruction of the k input fragments. We refer to the $k+r$ fragments as a *stripe*. To protect against failures, we store a stripe's fragments on independent disks. For efficiency all the fragments associated with a single blob are stored on the same set of $k+r$ disks. Further, all the fragments of the blob on a single disk are stored contiguously in a single file, called a *stripe stack*. While any k stripe stacks are sufficient to regenerate a blob, the current policy is to read all the $k+r$ stripe stacks on each request for the blob to allow each stripe stack to be integrity checked to prevent data corruption. Minimally we would like to ensure that the $k+r$ drives that store a blob can be concurrently spun up and accessed.

In the current Pelican prototype we are using $k = 15$ and $r = 3$. First, this provides good data durability with the anticipated disk and tray annual failure rates (AFRs) with a reasonable capacity overhead of 20%. The $r = 3$ allows up to three concurrent disk failures without data loss. Secondly, during a single blob read Pelican would like saturate a 10 Gbps network link, and reading from 15 disks concurrently can be done at approximately 1,500 MB/s or 12 Gbps provided that all the disks are spun-up and are chosen to respect the PCI bandwidth constraints. Thirdly, it is important that each blob is mapped to 18 domain-disjoint disks. If the disks cannot be spinning concurrently to stream out the data then in-memory buffering at the servers proportional to the size of the blob being accessed would be needed. If the disks are domain-disjoint then at most $k+r$ 128 kB fragments need to be buffered in memory for each read, possibly reconstructed, and then sent to the client. Similarly for writes only the fragments of the next stripe in the sequence are buffered in memory, the redundancy fragments are generated, and all fragments sent to disk.

The objective of the data layout algorithm is to maximize the number of requests that can be concurrently serviced while operating within the constraints. Intuitively, there is a queue of incoming requests, and each request has a set of disks that need to be spinning to service the request. We would like to maximize the probability that, given one executing request, we can find a queued request which can be serviced concurrently.

To understand how we achieve layout we first extend the definition of domain-conflict and domain-disjointness to sets of disks as follows: two sets, s_a and s_b are domain-conflicting if *any* disk in s_a is domain conflicted with *any* disk in s_b. Operations on domain-conflicting sets of disks need to be executed sequentially, while operations on domain-disjoint sets can be executed concurrently.

Imagine a straw-man algorithm in which the set of disks is selected with a simple greedy algorithm: all 1,152 disks are put into a list, one is randomly selected and all drives that have a domain-conflict with it are removed from the list, and this repeats until 18 disks have been chosen. This is very simple, but yields very poor concurrency. If you take two groups, a and b, of size g populated using this algorithm then each disk in b has a probability proportional to g of conflict with group a. Therefore, the probability for a and b to be domain-conflicting is proportional to g^2.

The challenge with more complex data layout is to minimize the probability of domain conflicts while taming the computational complexity of determining the set of disks to be used to store a blob. The number of combinations of 18 out of 1,152 disks, C_{18}^{1152}, is large.

In order to handle the complexity we divided the disks into l groups, such that each disk is a member of a single group and all disks within a group are domain-disjoint, so they can be spinning concurrently. The group abstraction then removes the need to consider individual disks

[1] Non-systematic codes could also be used if beneficial [3].

or constraints. The complexity is now determining if l^2 pairs of logical groups are domain-disjointed or not, rather than C_{18}^{1152} sets of disks.

To improve concurrency compared to the straw-man algorithm, we enforce that if one disk in group a collides with group b, *all* the disks in a collide with b. In other words we make groups either fully colliding or fully disjoint, which reduces the collision probability from being proportional to g^2 to being proportional to g. This is close to the lower bound on the domain-collision probability for the Pelican hardware because g is the number of cooling domains used by a group (only one disk can be active per cooling domain and disks within a group are domain-disjoint).

Figure 2 shows a simplified example of how we assign disks to groups. The black squares show one group of 12 disks, the red-and-white squares another group. Notice they collide in all their power and cooling domains and so both groups cannot be spinning simultaneously. We start with the black group and generate all its rotations, which defines 12 mutually-colliding groups: the light-blue squares. These groups all fully collide, and we call this set of groups a *class*. Within a class only one group can be spinning at a time because of the domain conflicts. However the remaining domains are not conflicted and so available to form other classes of groups which will not collide with any of the 12 groups in the first class. By forming classes of maximally-colliding groups we reduce collisions between the other remaining groups and so greatly improve the available concurrency in the system.

Selecting l is reasonably straightforward: we wish to maximize (to increase scheduling flexibility) the number l of groups of size g given $l \times g = 1152$ and with $g >= k + r$ (groups have to be large enough to store a stripe). In the current implementation we use $g = 24$ rather than $g = 18$ so that a blob can always entirely reside within a single group even after some disks have failed. Stripe stacks stored on failed drives are initially regenerated and stored on other drives in the group. Hence $l = 48$; the 48 groups divide into 4 classes of 12 groups, and each class is independent from the others.

Using groups for the data layout has several benefits: *(i)* groups encapsulate all the constraints because disks in a group are domain-disjointed by definition; *(ii)* groups define the concurrency that can be achieved while servicing a queue of requests: groups from the same class have disks that share domain-conflicts and so need to be serviced sequentially, while groups from different classes have disks that are all domain-disjoint so can be serviced concurrently; *(iii)* groups span multiple failure domains: they contain disks distributed across the trays and all backplanes; and *(iv)* groups reduce time required to recover from a failed disk because all the required data is contained within the group.

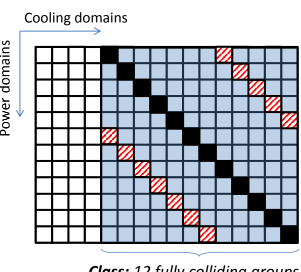


Figure 2: Two fully colliding groups.

A blob is written to $k + r$ disks in a single randomly selected group. The group has 24 disks but the blob only needs 18 disks. To select the disks to store the blob we split the group's disks into six sets each containing disks from the same backplane failure domain. The six sets are then ordered on spare capacity and the three disks with the highest spare capacity are selected. As we will show this simple approach achieves a high disk utilization.

Pelican preserves the group arrangement even when disks fail. A disk failure triggers rebuilds of all the blobs that stored a stripe stack on it. Rebuilding a single blob requires reading at least k stripe stacks and regenerating the missing stripe stack(s) across other disks in the group. By requiring blobs to be entirely within a group we ensure that rebuild operations use disks which are all spinning concurrently. The alternative would require buffering data in memory and spinning disks up and down during rebuild, which would slow down rebuild.

We use an off-rack metadata service called the *catalog* which is durable and highly available. Once disks have been chosen for a write request, the catalog is updated. It holds the mapping from a blob key to the 18 disks and group which store the blob, and other ancillary metadata. The catalog is modified during write, rebuild and delete requests, and information is looked up during read requests.

Using groups to abstract away the underlying hardware constraints is an important simplification for the IO scheduler: it needs simply consider which class the group is in rather than the constraints on all the drives. As we showed, increasing the number of groups which totally collide also increases the number of independent groups leading to better throughput and lower latency for operations. In the next section, we describe the IO scheduler in detail.

2.2.3 IO scheduler

In Pelican spin up is the new seek latency. Traditional disk schedulers have been optimized to re-order IOs in order to minimize *seek* latency overheads [14, 16]. In

Pelican we need to re-order requests in order to minimize the impact of *spin up* latency. The four classes defined by the data layout are domain-disjoint and are thus serviced independently. For each class, we run an independent instance of the scheduler that only services requests for its class. It has no visibility of requests on other classes and therefore the re-ordering happens at a class-level.

Traditional IO re-ordering attempts to order the requests to minimize the disk's physical head movement; every IO in the queue has a *different* relative cost to every other IO in the queue. We can define a cost function $c_d(h_l, IO_1, IO_2)$ which, given two IO requests IO_1 and IO_2 and the expected head position h_l, calculates the expected time cost of servicing IO_1 and then IO_2. This cost function will take into account the location of data being read or written, the rotational speed of the disk platters and the speed at which the head can move. This is a continuous function and $c_d(h_l, IO_1, IO_2) \neq c_d(h_l, IO_2, IO_1)$. In contrast, in Pelican there is a fixed constant cost of spinning up a group, which is independent of the current set of disks spinning. The cost function is $c_p(g_a, IO_g)$ where g_a is the currently spinning group and an IO has a group associated with it, so IO_g refers to an IO operation on group g. The cost function is binary, and if $g_a = g$ then the cost is zero, else it is 1.

We define the cost c of a proposed schedule of IO request as the sum of $c_p()$ for each request, assuming that the g of the previous request in the queue is g_a for the next request. Only one group out of the 12 can be spun up at any given time, and if there are q requests queued, and we use a FIFO queue, then $c \approx 0.92q$, as there is a probability of 0.92 that two consecutive operations will be on different groups. Note that there is an upper bound $c = q$, where every request causes a group to spin up.

The goal of the IO scheduler is to try to minimize c, given some constraint on the re-ordering queuing delay each request can tolerate. When $c \approx q$ then Pelican yields low throughput, as each spin up incurs a latency of at least 8 seconds. This means that in the *best case* only approximately 8 requests are serviced *per minute*. This also has the impact that, not only is throughput low, but the queuing latency for each operation will be high, as requests will be queued during the spin ups for the earlier requests in the queue.

Another challenge is ensuring that the window of vulnerability post-failures is controlled to ensure the probability of data loss is low. A disk failure in a group triggers a set of rebuild operations to regenerate the lost stripe-stacks. This rebuild requires activity on the group for a length of time equal to the data on the failed disk divided by the disk throughput (e.g., 100 MBps) since $k + r - 1$ stripe-stack reads and 1 stripe-stack write proceed concurrently. During the rebuild Pelican needs to service other requests for data from the affected group as well as other groups in the same class. Simply prioritizing rebuild traffic over other requests would provide the smallest window of vulnerability, but would also cause starvation for the other groups in the class.

The IO scheduler addresses these two challenges using two mechanisms: *request reordering* and *rate limiting*. Internally each scheduler instance uses two queues, one for rebuild operations the other for all other operations. We now describe these two mechanisms.

Reordering. In each queue, the scheduler can reorder operations independently. The goal of reordering is to batch sets of operations for the same group to amortize the group spin up latency over the set of operations. Making the batch sizes as large as possible minimizes c, but increases the queuing delay for some operations. To quantify the acceptable delay we use the delay compared to FIFO order.

Conceptually, the queue has a timestamp counter t that is incremented each time an operation is queued. When an operation r is to be inserted into the queue it is tagged with a timestamp $t_r = t$ and is assigned a reordering counter $o_r = t$. In general, $o_r - t_r$ represents the absolute change in ordering compared to a FIFO queue. There is an upper bound u on the tolerated re-ordering, and $o_a - t_a \leq u$ must hold for all operations a in the queue. The scheduler examines the queue and finds l, the last operation in the same group as r. If no such operation exists, r is appended to the tail of the queue and the process completes. Otherwise, the scheduler performs a check to quantify the impact if r were inserted after l in the queue. It considers all operations i following l. If $o_i + 1 - t_i \leq u$ no longer holds for any i, then r is appended to the tail of the queue. Otherwise all o_i counters are incremented by one, and r is inserted after l with $o_r = t_r - |i|$ where $|i|$ is the number of requests i, which r has overtaken.

To ease understanding of the algorithm, we have described the u in terms of the number of operations, which works if all the operations are for a uniform blob size. In order to support non-uniform blob sizes, we operate in wall clock time, and estimate dynamically the time each operation will be serviced, given the number of group spin ups and the volume of data to be read or written by operations before it in the queue. This allows us to specify u in terms of wall clock time.

This process is greedily repeated for each queued request, and guarantees that: *(i)* batching is maximized unless it violates fairness for some requests; and *(ii)* for each request the reordering bound is enforced. The algorithm expresses the tradeoff between throughput and fairness using u, which controls the reordering. For example, setting $u = 0$ results in a FIFO service order providing fairness, conversely setting $u = \infty$ minimizes the number of spin ups which increases throughput, but also means the request queuing delay is unbounded.

Rate limiting. The rate at which each queue is serviced is then controlled, to manage the interference between the rebuild and other operations. In particular, we want to make sure that the rebuild traffic gets sufficient resources to allow it to complete the rebuild within an upper time bound, to allow us to probabilistically ensure data durability if we know the AFR rates of the hardware.

The scheduler maintains two queues, one for the rebuild operations and one for other operations. We use a weighted fair queuing mechanism [6, 12] across the two queues which allows us to control the fraction of resources dedicated to servicing the rebuild traffic.

The approximate time to repair after a single disk failure is $x/t \times 1/w$ where x is the amount of data on the failed disk, t is the average throughput of a single disk (e.g., 100 MB/s) and w is the fraction of the resources the scheduler allocates to the rebuild.

3 Implementation

Early experience with the hardware has highlighted a number of other issues with right-provisioning. The first is to ensure that we do not violate the cooling or power constraints from when power is applied until the OS has finished booting and the Pelican service is managing the disks. We achieve this by ensuring the disks do not spin up when first powered, as done by RAID enclosures. Our first approach was to float pin 11 of the SATA power connector, which means that the drive spins up only when it successfully negotiates a PHY link with the HBA. We modified the Windows Server disk device driver to keep the PHY disabled until the Pelican service explicitly enables it. Once enabled the device driver announces it to the OS as normal, and Pelican can start to use it. However, this added considerable complexity, and so we moved to use Power Up In Standby (PUIS) where the drives establish a PHY link on power up, but require an explicit SATA command to spin up. This ensures disks do not spin without the Pelican storage stack managing the constraints.

At boot, once the Pelican storage stack controls the drives, it needs to mount and check every drive. Sequentially bring each disk up would adhere to the power and cooling constraints, but provides long boot times. In order to parallelize the boot process we exploit the group abstraction. We generate an initialization request for each group and schedule these as the first operation performed on each group. We know that the disks within the groups are domain-disjoint, so we can perform initialization concurrently on all 24 disks in the group. If there are no user requests present then four groups (96 disks) are concurrently initialized, initializing the entire rack takes the time required to sequentially initialize 12 disks (less than 3 minutes). External read and write requests can be concurrently scheduled, with the IO scheduler effectively on-demand initializing groups, amortizing the spin up time. The first time a disk is seen, Pelican writes a fresh GPT partition table and formats it with an NTFS filesystem.

During early development we observed unexpected spin ups of disks. When Pelican spins down a disk, it drains down IOs to the disk, unmounts the filesystem, and then sets the OFFLINE disk attribute. It then issues a STANDBY IMMEDIATE command to the disk, causing it to spin down. Disks would be spun up without a request from Pelican, due to services like the SMART disk failure predictor polling the disk. We therefore added a special *No Access* flag to the Windows Server driver stack that causes all IOs issued to a disk marked to return with a “media not ready” error code.

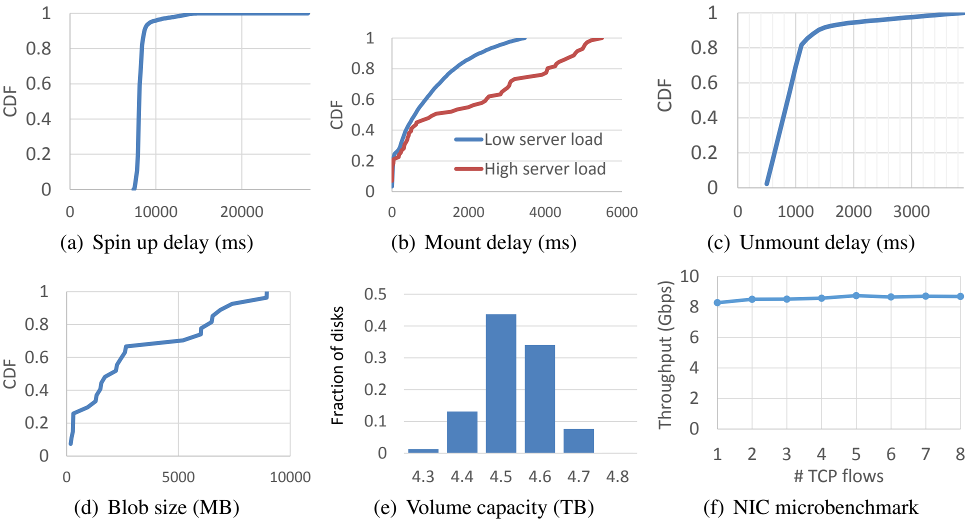
We also experienced problems with having many HBAs attached to the PCIe bus. The BIOS is responsible for initial PCI resource allocations of bus numbers, memory windows, and IO port ranges. Though neither the HBAs nor the OS require any IO ports, a small number are exposed by the HBA for legacy purposes. PCI requires that bridges decode IO ports at a granularity of 4 kB and the total IO port space is only 64 kB. We saw problems with BIOS code hanging once the total requirements exceeded 64 kB instead of leaving the PCI decode registers disabled and continuing. We have a modified BIOS on the server we use to ensure it can handle all 72 HBAs.

4 Evaluation

This section evaluates Pelican and in particular quantifies the impact of the resource right-provisioning on performance. We have a prototype Pelican rack with 6 chassis and a total of 1,152 disks. Our prototype uses archival class disks from a major disk manufacturer. Six PCIe uplinks from the chassis backplanes are connected to a single server using a Pelican PCIe aggregator card. The server is an HP ProLiant DL360p Gen8 with two eight-core Intel Xeon E5-2665 2.4GHz processors, 16 GB DRAM, and runs Windows Server 2012 R2. The server has a single 10 Gbps NIC. In order to allow us to evaluate the final Pelican configuration with two servers and to compare to alternative design points, we have developed a discrete event-based Pelican simulator. The simulator runs the same algorithms and has been cross-validated against the storage stack running on the full rack.

4.1 Pelican Simulator

The discrete event simulator models the disks, network and PCIe/SATA physical topology. In order to ensure



(a) Spin up delay (ms) (b) Mount delay (ms) (c) Unmount delay (ms)

(d) Blob size (MB) (e) Volume capacity (TB) (f) NIC microbenchmark

Figure 3: Distributions and parameters used in the simulation.

that the simulator is accurate we have parameterized the simulator using micro-benchmarks from the rack. We have then also cross-validated the simulator against the rack for a range of workloads.

We measure the spin up delays (shown in Figure 3(a)) and delays for mounting and unmounting a drive once it is spun up (shown in Figure 3(b) and 3(c)). For the spin up and unmounts delays we spun up and down disks 100,000 times, and measured the spin up and unmount latency. We observe that the volume mount delays are dependent on the load of the system. Under heavy load, the probability that at least one disk in a group is a straggler when all disks in the group are being mounted is much higher than when under low load. We therefore generate the mount delay distributions by taking samples of mount latencies during different load regimes on the Pelican. Figure 3(b) compares the distributions taken during high and low loads. In simulation, the mount delays are sampled from the distribution that corresponds to the current load.

For the disk throughput we measured the actual throughput of the disks in a quiescent system, and configure the simulator with an average disk throughput of 105 MB/s. Seeks are simulated by including a constant latency of 4.2ms for all disk accesses as specified in the disk data-sheet. In Pelican, seek latency has negligible impact on performance. Reads are for large blobs, and even though they are striped over multiple disks, each stripe stack will be a single contiguous file on disk. Further, as we can see from Figure 3(a) and 3(b) the latency of spinning the disk up and mounting the volume heavily dominate over seek time. The simulator uses a distribution of disk sizes shown in Figure 3(e). The capacities shown are for the drive capacity after formatting with NTFS. Finally, the Pelican is not CPU bound so we do not model CPU overheads.

In order to allow us to understand the performance effects of right-provisioning in Pelican, we also simulate a system organized like Pelican but with full provisioning for power and cooling which we denote as *FP*. In the FP configuration disks are *never spun down*, but the same physical internal topology is used. The disks are, as with Pelican, partitioned into 48 groups of 24 disks, however, in the FP configuration all 48 groups can be concurrently accessed. There is no spin up overhead to minimize, so FP maintains a queue per group and services each queue independently in FIFO order. FP represents an idealized configuration with no resource constraints and is hard to realize in practice. In particular, due to the high disk density in Pelican, little physical space is left for cooling and so forth.

In both configurations we assume, since the stripe stacks are contiguous on a disk, that they can be read at full throughput once the disk is spun up and mounted. The simulator models the PCIe/SATA and network bandwidth at flow level. For congested links the throughput of each flow is determined by using max-min fair sharing. As we will show in Section 4.4 we are able to cross-validate the Pelican simulator with the Pelican rack with a high degree of accuracy.

4.2 Configuration parameters

In all experiments the Pelican rack and simulator are configured with 48 groups of 24 disks as described. The groups are divided into 4 classes, and a scheduler is used per class. Blobs are stored using a 15+3 erasure encoding so each blob has 15 data blocks and 3 redundancy

blocks, all stored on separate disks. The maximum queue depth per scheduler is set to 1000 requests, and the maximum reordering allowance is set to 500 GB. For the cross-validation configuration, all 4 schedulers are run by a single server with one 10 Gbps NIC. In the rest of the evaluation, the rack has two servers such that each server runs two schedulers. Each server is configured with 20 Gbps network bandwidth, providing an aggregate network throughput of 40 Gbps for the entire Pelican rack.

4.3 Workload

We expect Pelican to be able to support a number of workloads, including archival workloads that would traditionally use tape. Pelican represents a new design point for cloud storage, with a small but non-negligible read latency and a limited aggregate throughput. We expect tiering algorithms and strategies will be developed that can identify cold data stored in the cloud that could be serviced by a Pelican. Due to the wide range of workloads we would like to support, we do not focus on a single workload but instead do a full parameter sweep over a range of possible workload characteristics. We generate a sequence of client read requests using a Poisson process with an average arrival rate $1/\lambda$, and vary $\lambda = 0.125$ to 16. For clarity, in the results we show the average request rate per second rather than the λ value. As write requests are offloaded to other storage tiers, we assume that servicing read requests is the key performance requirement for Pelican and, hence focus on read workloads. The read requests are randomly distributed across all the blobs stored in the rack. Unless otherwise stated requests operate on a blob size of 1 GB, to allow the metric of requests per second to be easily translated into an offered load. Some experiments use a distribution of blob sizes which is shown in Figure 3(d); a distribution of VHD image sizes from an enterprise with mean blob size of 3.3 GB.

In all simulator experiments, we wait for the system to reach steady state, then gather results over 24 simulated hours of execution.

4.4 Metrics

Our evaluation uses the following metrics:

Completion time. This is the time between a request being issued by a client and the last data byte being sent to the client. This captures the queuing delay, spin up latency and the time to read and transfer the data.

Time to first byte. The time between a request being issued by a client and the first data byte being sent to the client. This includes the queuing delay and any disk spin up and volume mount delays.

Service time. This is the time from when a request is dequeued by the scheduler to when the last byte associated with the request is transferred. This includes delays due to spinning up disks and the time taken to transfer the data, but excludes the queuing delay.

Average reject rate. These experiments use an open loop workload; when the offered load is higher than the Pelican or FP system can service, requests will be rejected once the schedulers' queues are full. This metric measures the average fraction of requests rejected. The NIC is the bottleneck at 5 requests per second for 40 Gbps. Therefore in all experiments, unless otherwise stated, we run the experiment to a rate of 8 request per second.

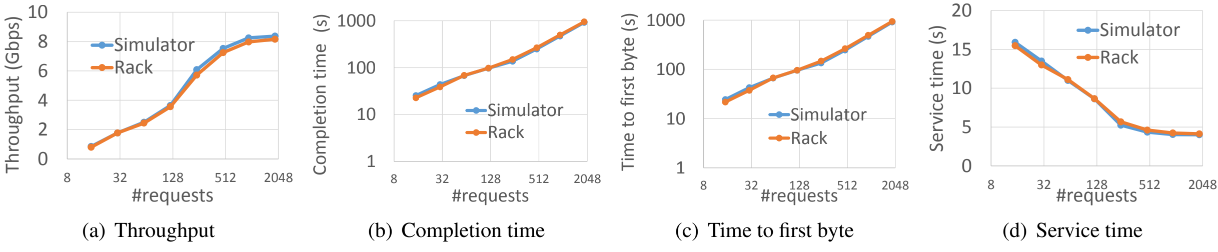
Throughput. This is the average rack network throughput which is calculated as the total number of bytes transferred during the experiment across the network link divided by the experiment duration.

First we cross-validated the simulator against the current Pelican rack. The current prototype rack can support only one disk spinning and one disk spinning up per tray, rather than two disks spinning up. The simulator is configured with this restriction for the cross-validation. The next revision of the rack and drives will be able to support two disks spinning up per tray. We configure the simulator to match the prototype Pelican rack hardware. A large number of experiments were run to test the algorithmic correctness of the simulator against the hardware implementation.

During the initial experiments on the real platform, we noticed that the NIC was unable to saturate the 10 Gbps NIC. Testing the performance of the NIC in isolation on an unloaded server using TTCP [10] identified that it had a peak throughput of only 8.5 Gbps. Figure 3(f) shows the network throughput as a function of the number of TCP flows. Despite our efforts to tune the NIC, the throughput of the NIC never exceeded 8.5 Gbps. For the cross-validation we configure the simulator to use an 8.5Gbps NIC.

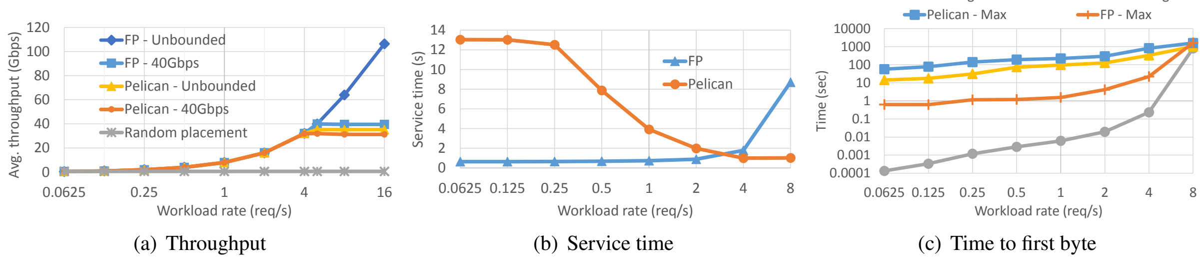
We ran a set of performance evaluation experiments where we generated a set of trace files consisting of a burst of b read requests for 1GB blobs uniformly distributed over 60 seconds, where we varied b from 15 to 1,920. We created a test harness that runs on a test server and reads a trace file and performs the read operations using the Pelican API running on the Pelican rack. Each experiment ran until all requests had been serviced. We replayed the same trace in the simulator. In both cases we pre-loaded the Pelican with the same set of blobs which are read during the trace.

To cross-validate we compare the mean throughput, request completion times, service times and times to first byte for different numbers of requests. The comparison is summarized in Figure 4. Figures 4(a) to 4(d) re-



(a) Throughput (b) Completion time (c) Time to first byte (d) Service time

Figure 4: Cross-validation of simulator and Pelican rack.



(a) Throughput (b) Service time (c) Time to first byte

Figure 5: Base performance of Pelican.

spectively show average throughput of the system, and average per-request completion time, time to first byte and service time as a function of the number of requests. These results show that the simulator accurately captures the performance of the real hardware for all the considered metrics. Traditionally, simulating hard disk-based storage accurately is very hard due to the complexity of the mechanical hard drives and their complex control software that runs on the drive [17]. In Pelican, the overheads are dominated by the spin up and disk mount latencies. The IO pattern at each disk is largely sequential, hence we do not need to accurately simulate the low-level performance of each physical disk.

All further results presented for Pelican and FP use this cross-validated simulator.

4.5 Base performance

The first set of experiments measure the base performance of Pelican and FP. Figure 5(a) shows the throughput versus the request rate for Pelican, FP and for the straw-man random layout (described in Section 2.2.2).

The straw-man performs poorly because the random placement means that the probability of being able to concurrently spin up two sets of disks to concurrently service two requests is very low. Across all request rates, it never achieves a throughput of more than 0.7 Gbps. The majority of requests are processed sequentially with group spin ups before each request. The latency of spinning up disks dominates and impacts throughput.

In Figure 5(a) we can also see that the throughput for

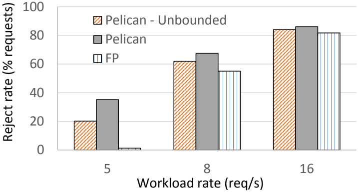


Figure 6: Reject rates versus workload request rate.

the default configurations of Pelican and FP configured with two servers each with 20 Gbps network bandwidth (labelled 40 Gbps in Figure 5(a)) is almost identical up to 4 requests per second. These results show that, for throughput, the impact of spinning up disks is small: across all the 40 Gbps configurations Pelican achieves a throughput within 20% of FP. The throughput plateaus for both systems after 5 requests per second at 40 Gbps which is the aggregate network bandwidth.

Figure 5(a) also shows the results for Pelican and FP configured with unbounded network bandwidth per server. This means the network bandwidth is never the bottleneck resource. This clearly shows the impact of right-provisioning. FP is able to utilize the extra resources in rack when the network bandwidth is no longer the bottleneck, and is able to sustain a throughput of 106 Gbps. In contrast, Pelican is unable to generate load for the extra network bandwidth because it is right-provisioned and configured for a target throughput of 40 Gbps.

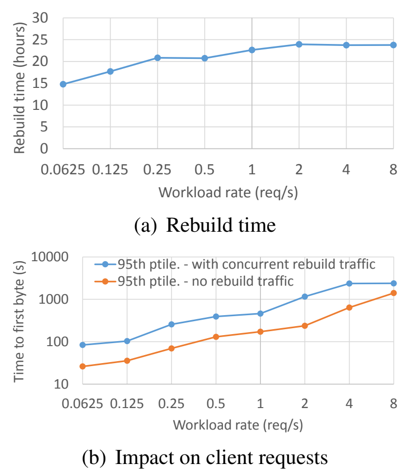
Figure 6 shows the average reject rates for all the configurations under different loads, except for FP with un-

bounded bandwidth as it never rejects a request. No configuration rejects a request for rates lower than 5 requests per second. At 5 requests per second, Pelican and Pelican unbounded start to become overloaded and reject 35% and 21% of the requests respectively. FP also marginally rejects about 1% of requests because request queues are nearly full and minor fluctuations in the Poisson arrival rate occasionally cause some requests to be rejected. As the request rates increase the rejection rate increase significantly, with more than half of the submitted requests rejected.

The results above show that Pelican can achieve a throughput comparable to FP. The next results explore the per request service time. Figure 5(b) shows the average request service time as a function of the request rate. For FP when the offered load is low, there is no contention on the network, and hence the service time is simply the time taken to read the blob from the disks. Hence, for 1 GB blobs the lower bound on service time is approximately 0.65 seconds, given the disk throughput of 105 MB/s. As the offered load increases, concurrent requests get bottlenecked on the network, leading to the per-request service time increasing.

For Pelican, request rates in the range of 0.0625 and 0.25 per second, results in most requests incurring a spin up delay because requests are uniformly distributed across groups, and there are only 4 groups spun up at any time out of 48, so the probability of a request being for a group already spinning is low. As the request rate increases, the probability of multiple requests being queued for same group increases, and the average service time decreases. Above 0.25 requests per second there is sufficient number of requests being queued for the scheduler to re-order them to reduce the number of group spin ups. Although requests will not be serviced in the order they arrive, the average service time decreases as multiple requests are serviced per group spin up. Interestingly, for a rate of 4 requests per second and higher the service time for Pelican drops below FP, because in FP 48 requests are serviced concurrently versus only 4 in Pelican. The Pelican requests therefore obtain a higher fraction of the network bandwidth per request and so complete in less time.

Next we explore the impact on data access latency of needing to spin up disks. Figure 5(c) shows the average and maximum time on a log scale to first byte as a function of request rate for Pelican and FP. Under low offered load the latency is dominated by the disk seek for FP and by spin up for Pelican, so FP is able to get the first byte back in tens of milliseconds on average, while Pelican has an average time to first byte of 14.2 seconds even when idle. The maximum times show that FP has a higher variance, due to a request that experiences even short queuing delay being impacted by the service time



(a) Rebuild time

(b) Impact on client requests

Figure 7: Scheduler performance.

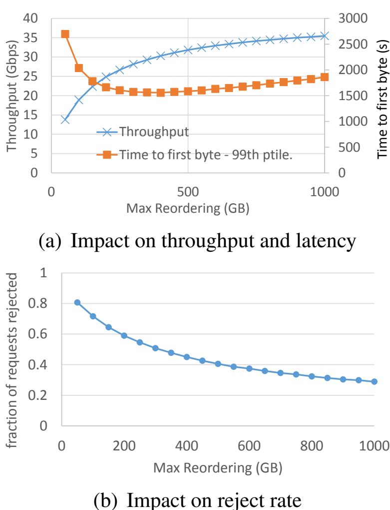
of requests scheduled before it. However, the maximum for FP is over an order of magnitude lower than the mean for Pelican, until the request rate is 5 requests per second and are bottlenecked on the network bandwidth.

Overall, the results in Figure 5 show that Pelican can provide a high utilization with reasonable latency. In an unloaded Pelican blobs can be served quickly, whereas under heavy load, high throughput is delivered.

4.5.1 Impact of disk failure

Next, we evaluate how recovering from a disk failure impacts the time to first byte for concurrent client requests. We also evaluate the time Pelican takes to rebuild all lost blobs. The experiment is the same as that of the previous section, except that we mark one disk as failed once the system reaches steady state. The disk contains 4.28 TB of data and 64,329 blobs stored in the group have a stripe stack on the failed disk. For each blob $\geq k$ undamaged stripe stacks are read and the regenerated stripe stack is written to the same group, which has sufficient spare capacity distributed across the other disks in the group. The scheduler rate-limits client requests to ensure that the rebuild has at least 50% of the throughput.

Figure 7(a) shows the time to rebuild and persist all the blobs versus the client request rate. At low client request rates the offered client load is low enough that it does not use all the resources available to it, and due to work conservation, the time to recover is low. As the offered client traffic increases, the time taken to recover from the disk failure grows. The rebuild time plateaus below 24 hours, when both the rebuild and client request use their allocated 50%. Figure 7(b) shows the time to first byte for the 95th percentile of client requests versus client request rate with and without rebuild requests. The rate limiting increases the 95th percentile time to first byte for



(a) Impact on throughput and latency

(b) Impact on reject rate

Figure 8: Cost of fairness.

client requests by a factor of 2.2 at high workload rates. The rebuild traffic only affects client requests in the class impacted by the disk failure. Other classes observe no performance impact.

4.6 Cost of fairness

The next experiments evaluate the impact on fairness by the Pelican scheduler. We fix the client workload at 5 read requests per second, the maximum rate which the network could sustain. We ran the experiment varying u, the tolerance on re-ordering, between 50 and 1000 GB. Figure 8(a) shows the throughput and 99th percentile of time to first byte as a function of u. We show the 99th percentile for time to first byte to quantify the impact on the tail, those requests most impacted by queuing delay.

Changing u from 50 to 350 GB nearly doubles the throughput of the system. Increasing u beyond this has only a modest impact on throughput. This is because the fraction of time spent spinning up drives is low therefore the majority of time is being spent reading data. When u is below 350 GB, the 99th percentile of time to first byte is dominated by the queuing delay. A significant fraction of the time is spent spinning disks up, impacting both throughput and latency of request. Hence increasing reordering improves throughput and reduces the queuing delay. At above 350GB the 99th percentile for time to first byte increases, despite the slight increase in throughput. This is slightly counter intuitive, until you remember that as u increases we are allowing requests to be queued for longer. This is then reflected when looking at the 99th percentile. This also impacts the number of requests that are rejected. Figure 8(b) shows the reject rate of the system as a function of u. Changing u from 50 to 1000 GB reduces the percentage of rejected requests from nearly 85% to approximately 25% due to increased throughput.

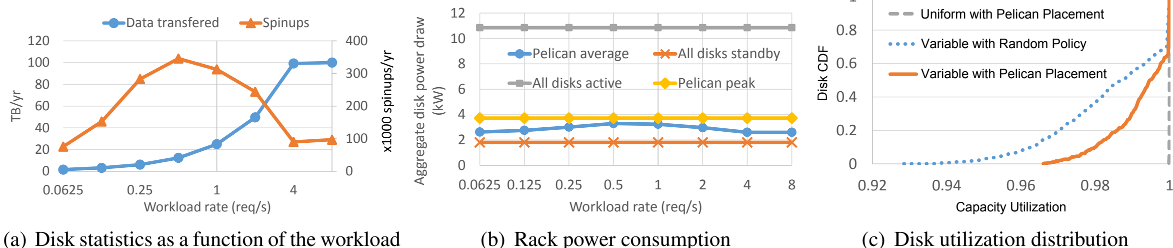
4.7 Disk Lifetime

An obvious concern is the impact on the lifetime of the disks of spinning them up and down. Also, the new class of archival drive that systems like Pelican use are rated for a number of terabytes read and written per year. We therefore ran an experiment to determine the average number of spin ups per year as well as the expected number of terabytes transferred. Figure 9(a) shows the average number of spin ups and terabytes transferred per year as a function of the workload rate. The average data transferred increases with the request rate and peaks at 99 TB per year when the request rate saturates the system. This is within the specification for the new generation of archival drives. Currently in the prototype Pelican we do not do any proactive background data scrubbing, which is common in storage systems to check for integrity issues. Background scrubbing increases the volume of data transferred per disk, which in itself impacts the disk AFR. We are currently long-term empirically testing the drives and plan to add scrubbing functionality once we have a better understanding of the drives longer term performance.

The number of spin ups is also shown in Figure 9(a); interestingly the peak is at 0.5 requests per second. Below this rate the number of spin ups grows with load as the scheduler has little opportunity to reorder requests. Above this rate the number of spin ups decreases because the queues are long enough for the scheduler to perform reordering. At 4 requests per second the scheduler hits the maximum reordering limit and the number of spin ups remains constant as the request rate increases further. We believe that the archival class of drive that we are using can tolerate this many spin up cycles per year with minimal impact on the AFR. It should be noted that these are controlled head park and unpark operations triggered by issuing a SATA command to the drive, which is then allowed to park the head. They are not induced by sudden power failure. When a disk is spun down, all the electronics in the drive are still powered and operating.

4.8 Power Consumption

We now quantify the power savings resulting from power right-provisioning. The prototype Pelican hardware enables us to measure the power draw of each tray independently. We ran an experiment in which we sequentially spun up then down every disk in a tray, sampling the tray power draw. This allows us to estimate the average power draw of a disk in standby, spinning up and active states. The tray power is dominated by the disks, so



(a) Disk statistics as a function of the workload

(b) Rack power consumption

(c) Disk utilization distribution

Figure 9: Power draw and disk usage.

we place all 16 disks in standby and estimate the power draw per disk as $\frac{1}{16}$th of the power draw of a tray. The average power draw for a disk active or spinning up is then computed using the average power draw of a tray with one disk in that test state minus the power draw of the other disks in standby. The power draws for the three disks states is: $P_{standby} = 1.56$ W, $P_{spinup} = 21.53$ W and $P_{active} = 9.42$ W.

We parameterize the simulator with these values and measure the power consumed by just the disks. Figure 9(b) shows the power draw of the 1,152 disks under different configurations as a function of the workload rate. The figure shows the average and peak power draw for Pelican. It also shows the lower bound when all drives in spun down in standby and, in order to allow comparison with a fully provisioned system, it shows the power draw for all drives spun up and active.

The average power draw of Pelican varies with the workload rate. The highest average power consumption which is 3.3 kW is reached when the number of spin ups is the highest, at around 0.5 requests per second because spin ups have the highest power draw. At this rate, there are sufficient requests to require frequent group spin ups, but not enough for extensive batching. For both lower and higher request rates, group spin ups are less frequent and the power consumption is close to 2.6 kW. Across all request rates the average Pelican power draw is between 3.6 and 4.1 times lower compared to the fully provisioned power draw with all disk spinning. Pelican peak power consumption is 3.7 kW and is reached when 96 drives are concurrently spinning up. For comparison, peak power draw is 3 times lower than all disks active. In the fully provisioned rack the peak power draw would be achieved if all 1,152 drives were concurrently spun up and would peak at 25.8kW. Of course, in practice some form of deferred spin up technique would be used.

4.9 Capacity Utilization

The final set of experiments evaluate the Pelican data layout algorithm with respect to capacity utilization, particularly with non-uniform disk capacities. We ran an experiment with a 100% write workload that stopped when the first write request was rejected because no group had sufficient remaining capacity to store the blob. We measure the per-disk utilization across the rack. For this experiment, the blobs sizes are selected using a distribution of VHD sizes from an internal company VHD store, with sizes from 200 MB to 9 GB with an average of 3.3 GB.

Figure 9(c) shows the CDF of per-disk utilization for three configurations. The solid line uses the Pelican placement policy with variable disk capacities, the dotted line is if the disks within a group are selected randomly from the subset that have sufficient spare capacity. Finally, the dashed line is for disks of equal size (set to the mean capacity of the disks used for variable). Comparing the Pelican approach to the random policy with variable capacity disks shows the benefit of Pelican approach.

To understand this more consider the rack utilization, rather than per-disk utilization. The total capacity utilization for an entire rack is 99.386% for Pelican with variable disk capacities as shown in figure 3(e) versus 99.998% for uniform disk capacities. At 1,152 disks, this implies that the equivalent of 7.07 disks are completely empty when using variable capacities when the first request is rejected. However, a traditional RAID system would have clipped all these drives to the minimum 4.2 TB. Therefore, compared to a 100% utilization at 4.2 TB, adapting to variable capacity is giving us $99.4 \times 4.55/4.2 = 107.7\%$ utilization.

5 Related Work

There has been extensive work on enabling disks to spin down under low load, including [5, 9, 2, 15]. Several of these systems proposed to modify the individual disk IO handling to increase periods of disk inactivity, allowing disks to spin down to save power [9, 13]. Write offloading [9] proposed allowing active disks to be used to buffer writes that were targeted against disks that were spun down. Only read requests for data not available required disks to be spun up, increasing the fraction of time disks could be spun down. Write-offloading makes no assumptions on the data layout used and worked at

the block level. Pergamum [13] is a MAID-like system; it incorporates NVRAM to handle meta-data, absorbing meta-data reads and writes as well as buffering writes to spun down disks. It also uses spin up tokens to limit peak power draw. These mechanisms could be exploited to support right-provisioning of power. However, unlike Pelican, Pergamum allows clients to select the disks being used to store their data. In Pelican the data layout is a function of the physical location of the disks which determines their power and cooling domains, and Pelican schedules these accesses appropriately.

Massive Arrays Of Idle Disks (MAID) [5] systems assume that the power and cooling resources are provisioned to support a peak performance. In contrast, Pelican is right-provisioned for the workload rather than for peak hardware performance.

Systems such as Rabbit [2] and Sierra [15] are power-proportional. In a power-proportional system the power consumption is proportional to the load. This is usually achieved through careful data layout schemes [15]. However, again most of these systems assume at peak performance all disks can be spun up. In many ways Pelican is also power-proportional, the number of disks spinning is a function of the workload, except the number of drives spinning never exceeds 8% of the disks.

Pergamum [13] also assumed a model where each disk was effectively connected directly to the network using an Ethernet port, with a small processor board mounted in front of each disk for local processing. This general model has been adopted by the Seagate Kinetic drives [11]. Pelican uses standard SATA archival drives and PCIe at the rack-scale, primarily to minimize costs. Further, this allows a single server to accurately control the drives and their states during boot time and during operation.

Finally, Amazon Glacier [1] and the Facebook cold data storage SKU [8] have never had public details released on their software stack to date. Facebook has released a design of the hardware through the Open Compute Project (OCP). Compared to the public information, Pelican has a higher disk density, lower system power consumption per disk, lower hardware cost, and provides smaller failure domains when compared to the OCP Cold Storage reference design.

6 Future Work

Pelican raises a number of interesting questions which we leave open for future work. The current Pelican software stack was co-designed with the hardware. This has the benefit that we are able to right-provision Pelican, but it has the drawback that the disk group assignment is very brittle with respect to hardware changes. For example, changing the cooling or power domains, or adding a new constraint, would require a redesign of the data layout and re-working of the IO scheduler. Further, designing the storage stack to work within the constraints is not trivial and took many months, and getting it wrong is not always immediately or trivially visible, for example violating the vibration domain. We do not know if we have an optimal design, simply we have one that seems to perform well. To address these issues we are currently developing tools to automatically synthesize the data layout and IO scheduling policies. We believe that we can encapsulate the underlying principles that we learnt when building this storage stack in a tool. This will enable rapid (automatic) optimisation of cold storage.

The other issue is that for many years the enterprise storage community has avoided spinning disks up and down, as it tends to yield higher failure rates, and even worse, correlated failures. We hope to gain understanding of this empirically over time. In particular, the spinning up and down of groups together, while helping improve performance when batching multiple requests, means all drives in the same group have an identical history. If we observe correlated failures, we can be more conservative and spin up only the 18 disks which are required to service a particular operation. We can also make sure that all groups spin up with a particular minimum frequency, as well as watch particular performance metrics for early signs of high wear. Many of these things we will only discover over time and we look forward to reporting them to the community.

7 Conclusion

Pelican is designed to support workloads where data stored is rarely read, often referred to as cold data. Pelican is unique in that the hardware has been designed to be right-provisioned. In this paper we have described and evaluated how these hardware limitations impact the data layout and IO scheduling. We have shown that the Pelican mechanisms are effective and compared against a fully provisioned system.

Acknowledgements

We would like to thank the following for their help: Cheng Huang, Brad Calder, Thierry Fevrier, Karan Mehra, Erik Hortsch and David Goebel. We'd like to thank our shepherd, Emin Gün Sirer, for his comments and support. We'd also like to thank the anonymous reviews for their insightful and helpful feedback.

References

[1] Amazon glacier. http://aws.amazon.com/glacier/, August 2012.

[2] AMUR, H., CIPAR, J., GUPTA, V., GANGER, G. R., KOZUCH, M. A., AND SCHWAN, K. Robust and flexible power-proportional storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 217–228.

[3] ANDRÉ, F., KERMARREC, A.-M., MERRER, E. L., SCOUARNEC, N. L., STRAUB, G., AND VAN KEMPEN, A. Archiving cold data in warehouses with clustered network coding. In *Proceedings of EuroSys* (New York, NY, USA, Apr 2014), ACM.

[4] BLÖMER, J., KALFANE, M., KARP, R., KARPINSKI, M., LUBY, M., AND ZUCKERMAN, D. An XOR-Based Erasure-Resilient Coding Scheme. Tech. Rep. ICSI TR-95-048, University of California, Berkeley, August 1995.

[5] COLARELLI, D., AND GRUMWALD, D. Massive Arrays of Idle Disks for Storage Archives. In *IEEE Supercomputing* (July 2002).

[6] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures & Protocols* (New York, NY, USA, 1989), SIGCOMM '89, ACM, pp. 1–12.

[7] MARCH, A. Storage pod 4.0: Direct wire drives - faster, simpler, and less expensive. http://blog.backblaze.com/2014/03/19/backblaze-storage-pod-4/, March 2014.

[8] MORGAN, T. P. Facebook loads up innovative cold storage datacenter. http://www.enterprisetech.com/2013/10/25/facebook-loads-innovative-cold-storage-datacenter/, October 2013.

[9] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: Practical power management for enterprise storage. *Trans. Storage 4*, 3 (Nov. 2008), 10:1–10:23.

[10] Ntttcp. http://gallery.technet.microsoft.com/NTttcp-Version-528-Now-f8b12769, July 2013.

[11] SEAGATE. The seagate kinetic open storage vision. http://tinyurl.com/noj7glm, August 2014.

[12] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 1995), SIGCOMM '95, ACM, pp. 231–242.

[13] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), USENIX Association, p. 1.

[14] TEOREY, T. J., AND PINKERTON, T. B. A comparative analysis of disk scheduling policies. In *Proceedings of the Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1971), SOSP '71, ACM, pp. 114–.

[15] THERESKA, E., DONNELLY, A., AND NARAYANAN, D. Sierra: Practical power-proportionality for data center storage. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 169–182.

[16] WORTHINGTON, B. L., GANGER, G. R., AND PATT, Y. N. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1994), SIGMETRICS '94, ACM, pp. 241–251.

[17] WORTHINGTON, B. L., GANGER, G. R., PATT, Y. N., AND WILKES, J. On-line extraction of scsi disk drive parameters. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1995), pp. 146–156.

A Self-Configurable Geo-Replicated Cloud Storage System

Masoud Saeida Ardekani[1,2] and Douglas B. Terry[3]

[1]Inria
[2]Sorbonne Universités, UPMC Univ Paris 06
[3]Microsoft Research Silicon Valley

Abstract

Reconfiguring a cloud storage system can improve its overall service. Tuba is a geo-replicated key-value store that automatically reconfigures its set of replicas while respecting application-defined constraints so that it adapts to changes in clients' locations or request rates. New replicas may be added, existing replicas moved, replicas upgraded from secondary to primary, and the update propagation between replicas adjusted. Tuba extends a commercial cloud-based service, Microsoft Azure Storage, with broad consistency choices (as in Bayou), consistency-based SLAs (as in Pileus), and a novel replication configuration service. Compared with a system that is statically configured, our evaluation shows that Tuba increases the reads that return strongly consistent data by 63%.

1 Introduction

Cloud storage systems can meet the demanding needs of their applications by dynamically selecting when and where data is replicated. An emerging model is to utilize a mix of strongly consistent primary replicas and eventually consistent secondary replicas. Applications either explicitly choose which replicas to access or let the storage system select replicas at run-time based on an application's consistency and performance requirements [15]. In either case, the configuration of the system significantly impacts the delivered level of service.

Configuration issues that must be addressed by cloud storage systems include: (i) where to put primary and secondary replicas, (ii) how many secondary replicas to deploy, and (iii) how frequently secondary replicas should synchronize with the primary replica. These choices are complicated by the fact that Internet users are located in different geographical locations with different time zones and access patterns. Moreover, systems must consider the growing legal, security, and cost constraints about replicating data in certain countries or avoiding replication in others.

For a stable user community, static configuration choices made by a system administrator may be acceptable. But many modern applications, like shopping, social networking, news, and gaming, not only have evolving world-wide users but also observe time-varying access patterns, either on a daily or seasonal basis. Thus, it is advantageous for the storage system to *automatically* adapt its configuration subject to application-specific and geo-political constraints.

Tuba is a geo-replicated key-value store based on Pileus [15]. It addresses the above challenges by configuring its replicas automatically and periodically. While clients try to maximize the utility of individual read operations, Tuba improves the overall utility of the storage system by automatically adapting to changes in access patterns and constraints. To this end, Tuba includes a configuration service that periodically receives from clients their consistency-based service level agreements (SLAs) along with their hit and miss ratios. This service then changes the locations of primary and secondary replicas to improve the overall delivered utility. A key property of Tuba is that both read and write operations can be executed in parallel with reconfiguration operations.

We have implemented Tuba as middleware on top of Microsoft Azure Storage (MAS) [3]. It extends MAS with broad consistency choices as in Bayou [14], and provides consistency-based SLAs like Pileus. Moreover, it leverages geo-replication for increased locality and availability. Our API is a minor extension to the MAS Blob Store API, thereby allowing existing Azure applications to use Tuba with little effort while experiencing the benefits of dynamic reconfiguration.

An experiment with clients distributed in datacenters (sites) around the world shows that reconfiguration every two hours increases the fraction of reads guaranteeing strong consistency from 33% to 54%. This confirms that automatic reconfiguration can yield substantial benefits which are realizable in practice.

The outline of the paper is as follows. We review Pileus and Tuba in Section 2. We look under the hood of Tuba's configuration service in Section 3. Section 4 describes execution modes of clients in Tuba. In Section 5, we explain implementation details of the system. Our evaluation results are presented in Section 6. We review related work in Section 7 and conclude the paper in Section 8.

2 System Overview

In this section, we first briefly explain features that Tuba inherits from Pileus. Since we do not cover all technical issues of Pileus, we encourage readers to read the original paper [15] for more detail. Then, we overview Tuba and its fundamental components, and how it extends the features of the Pileus system.

2.1 Tuba Features from Pileus

Storage systems cannot always provide rapid access to strongly consistent data because of the high network latency between geographical sites and diverse operational conditions. Clients are forced to select less ideal consistency/latency combinations in many cases. Pileus addresses this problem by allowing clients to declare their consistency and latency priorities via SLAs. Each SLA comprises several subSLAs, and each subSLA contains a desired consistency, latency and utility.

The utility of a subSLA indicates the value of the associated consistency/latency combination to the application and its users. Inside a SLA, higher-ranked subSLAs have higher utility than lower-ranked subSLAs. For example, consider the SLA shown in Figure 1. Read operations with strong consistency are assigned utility 1 as long as they complete in less than 50 ms. Otherwise, the application tolerates eventually consistent data and longer response times though the rewarded utility is very small (0.01). Pileus, when performing a read operation with a given SLA, attempts to maximize the delivered utility by meeting the highest-ranked subSLA possible.

The replication scheme in Pileus resembles that of other cloud storage systems. Like BigTable [4], each key-value store is horizontally partitioned by key-ranges into *tablets*, which serve as the granularity of replication. Tablets are replicated at an arbitrary collection of storage sites. Storage sites are either primary or secondary. All write operations are performed at the primary sites. Secondary sites periodically synchronize with the primary sites in order to receive updates.

Rank	Consistency	Latency(ms)	Utility
1	Strong	50	1
2	Eventual	1000	0.01

Figure 1: SLA Example

Depending on the desired consistency and latency as specified in an SLA, the network delays between clients and various replication sites, and the synchronization period between primary and secondary sites, the Pileus client library decides on the site to which a read operation is issued. Pileus provides six consistency choices that can be included in SLAs: (i) strong (ii) eventual (iii) read-my-writes (RMW) (iv) monotonic reads (v) bounded(t), and (vi) causal.

Consider again the SLA shown in Figure 1. A Pileus client reads the most recent data and *hits* the first subSLA as long as the round trip latency between that client and a primary site is less than 50ms. But, the first subSLA *misses* for clients with a round trip latency of more than 50ms to primary sites. For these clients, Pileus reads data from any replica site and hits the second subSLA.

Pileus helps developers find a suitable consistency/latency combination given a fixed configuration of tablets. Specifically, the locations of primary and secondary replication sites, the number of required secondary sites, and the synchronization period between secondary and primary sites need to be specified by system administrators manually. However, a worldwide distribution of users makes it extremely hard to find an optimal configuration where the overall utility of the system is maximized with a minimum cost. Tuba extends Pileus to specifically address this issue.

2.2 Tuba's New Features

The main goal of Tuba is to periodically improve the overall utility of the system while respecting replication and cost constraints. To this end, it extends Pileus with a configuration service (CS) delivering the following capabilities:

1. performing a reconfiguration periodically for different tablets, and
2. informing clients of the current configuration for different tablets.

We note that the above capabilities do not necessarily need to be collocated at the same service. Yet, we assume they are provided by the same service for the sake of simplicity.

In order for the CS to configure a tablet's replicas such that the overall utility increases, it must be aware of the way the tablet is being accessed globally. Therefore, all clients in the system periodically send their *observed latency* and the *hit and miss ratios* of their SLAs to the CS.

The observed latency is a set comprising the latency between a client (e.g., an application server) and different datacenters. The original Pileus system also requires clients to maintain this set. Since the observed latency between datacenters does not change very often, this set is only sent every couple of hours, or when it changes by more than a certain threshold.

Tuba clients also send their SLAs' hit and miss ratios periodically. It has been previously observed that placement algorithms with client workload information (such as the request rate) perform two to five times better than workload oblivious random algorithms [10]. Thus, every client records aggregate ratios of all hit and missed subSLAs for a sliding window of time, and sends them to the CS periodically. The CS then periodically (or upon receiving an explicit request) computes a new configuration such that the overall utility of the system is improved, all constraints are respected, and the cost of the migrating to and maintaining the new configuration remains below some threshold.

Once a new configuration is decided, one or more of the following operations are performed as the system changes to the new configuration: (i) changing the primary replica, (ii) adding or removing secondary replicas, and (iii) changing the synchronization periods between primary and secondary replicas. In the next section, we explain in more detail how the above operations are performed with minimal disruption to active clients.

3 Configuration Service (CS)

The CS is responsible for periodically improving the overall utility of the system by computing and applying new configurations. The CS selects a new configuration by first generating all reasonable replication scenarios that satisfy a list of defined constraints.

For each configuration possibility, it then computes the expected gained utility and the cost of reconfiguration. The new chosen configuration is the one that offers the highest utility-to-cost ratio. Once a new configuration is chosen, the CS executes the reconfiguration operations required for making a transition from the old configuration to the new one.

In the remaining of this section, we first explain the different types of constraints and the cost model used by the CS. Then, we introduce the algorithm behind the CS to compute a new configuration. Finally, we describe how the CS executes different reconfiguration operations to install the new configuration.

3.1 Constraints

Given the simple goal of maximizing utility, the CS would have a *greedy* nature: it would generally decide to add replicas. Hence, without constraints, the CS could ultimately replicate data in all available datacenters. To address this issue, a system administrator is able to define constraints for the system that the CS respects.

Through an abstract constraint class, Tuba allows constraints to be defined on any attribute of the system. For example, a constraint might disallow creating more than three secondary replicas or disallow a reconfiguration to happen if the total number of online users is greater than 1 million. Tuba abides by all defined constraints during every reconfiguration.

Several important constraints are currently implemented and ready for use including: (i) Geo-replication factor, (ii) Location, (iii) Synchronization period, and (iv) Cost.

With geo-replication constraints, the minimum and maximum number of replicas can be defined. For example, consider an online music store. Developers may set the maximum geo-replication factor of tablets containing less popular songs to one, and set the minimum geo-replication factor of a tablet containing top-ten best selling songs to three. Even if the storage cost is relatively small, limiting the replication factor may still be desirable due to the cost of communication between sites for replica synchronization.

Location constraints are able to explicitly force replication in certain sites or disallow them in others. For example, an online social network application can respond to security concerns of European citizens by allowing replication of their data only in Europe datacenters.

With the synchronization period constraint, application developers can impose bounds on how often a secondary replica synchronizes with a primary replica.

The last and perhaps most important constraint in Tuba is the cost constraint. As mentioned before, the CS picks a configuration with the greatest ratio

of gained utility over cost. With a cost constraint, application developers can indicate how much they are willing to pay (in terms of dollars) to switch to a new configuration. For instance, one possible configuration is to put secondary replicas in all available datacenters. While the gained utility for this configuration likely dominants all other possible configurations, the cost of this configuration may be unacceptably large. In the next section, we explain in more detail how these costs are computed in Tuba.

Should the system administrator neglect to impose any constraint, Tuba has two default constraints in order to avoid aggressive replication and to avoid frequent synchronization between replicas: (1) a lower bound for the synchronization period, and (2) an upper bound on the recurring cost of a configuration.

3.2 Cost Model

The CS considers the following costs for computing a new configuration:

- Storage: the cost of storing a tablet in a particular site.
- Read/Write Operation: the cost of performing read/write operations.
- Synchronization: the cost of synchronizing a secondary replica with a primary one.

The first two costs are computed precisely for a certain period of time, and the third cost is estimated based on read/write ratios.

Given the above categories, the cost of a primary replica is the sum of its storage and read/write operation costs, and the cost of a secondary replica is the sum of storage, synchronization, and read operation costs. Since Tuba uses batching for synchronization to a secondary replica and only sends the last write operation on an object in every synchronization cycle, the cost of a primary replica is usually greater than that of secondary replicas.

In addition to the above costs, the CS also considers the cost of creating a new replica; this cost is computed as one-time synchronization cost.

3.3 Selection

Potential new configurations are computed by the CS in the following three steps:

Ratios aggregation. Clients from the same geographical region usually have similar observed access latencies. Therefore, as long as they use the same SLAs, their hit and miss ratios can be aggregated to reduce the computation. We note that this phase does not necessarily need to be in the critical path, and aggregations can be done once clients send their ratios to the CS.

Rank	Consistency	Latency(ms)	Utility
1	Strong	100	1
2	RMW	100	0.9
3	Eventual	1000	0.01

Figure 2: SLA of a Social Network Application

Configuration computation. In this phase, possible configurations that can improve the overall utility of the system are generated and sorted. For each missed subSLA, and depending on its consistency, the CS may produce several potential configurations along with their corresponding reconfiguration operations. For instance, for a missed subSLA with strong consistency, two scenarios would be: (i) creating a new replica near the client and making it the solo primary replica, or (ii) adding a new primary replica near the client and making the system run in multi-primary mode.

Each new configuration has an associated cost of applying and maintaining it for a certain period of time. The CS also computes the overall gained utility of every new configuration that it considers. Finally, the CS sorts all potential configurations based on their gained utility over their cost.

Constraints satisfaction. Configurations that cannot satisfy all specified constraints are eliminated from consideration. Constraint classes also have the ability to add configurations being considered. For instance, the minimum geo-replication constraint might remove low-replica configurations and create several new ones with additional secondary replicas at different locations.

3.4 Operations

Once a new configuration is selected, the CS executes a set of reconfiguration operations to transform the system from the current configuration. In this section, we explain various reconfiguration operations and how they are executed abstractly by the CS, leaving the implementation specifics to Section 5.

3.4.1 Adjust the Synchronization Period

When a secondary replica is added to the system for a particular tablet, a default synchronization period is set, which defines how often a secondary replica synchronizes with (i.e., receives updates from) the primary replica. Although this value does not affect

the latency of read operations with strong or eventual consistency, the average latency of reads with intermediary consistencies (i.e., RMW, monotonic reads, bounded, and causal) can depend heavily on the frequency of synchronization. Typically, the cost of adjusting the synchronization period is smaller than the cost of adding a secondary replica or of changing the locations of primary/secondary replicas. Hence, it is likely that the CS will decide to decrease this period to increase the hit ratios of subSLAs with intermediary consistencies.

For example, consider a social network application with the majority of users located in Brazil and India accessing a storage system with a primary replica located in Brazil, initially, and a secondary replica placed in South Asia with the synchronization period set to 10 seconds. Assume that the SLA shown in Figure 2 is set for all read operations. Given the fact that the round trip latency between India and Brazil is more than 350 ms, the first subSLA will never hit for Indian users. Yet, depending on the synchronization period and frequency of write operations performed by Indian users, the second subSLA might hit. Thus, if the CS detects low utility for Indian users, a possible new configuration would be similar to the old one but with a reduced synchronization period.

In this case, the chosen operation to apply the new configuration is *adjust_sync_period*. Executing this operation is very simple since the value of the synchronization period need only be changed in the secondary replica. Clients do not directly observe any difference between the new configuration and the old one, but they benefit from a more up-to-date secondary replica.

3.4.2 Add/Remove Secondary Replica

In certain cases, the CS might decide to add a secondary replica to the system. For example, consider an online multiplayer game with the SLA shown in Figure 3 and where the primary replica is located in the East US region. In order to deliver a better user experience to gamers around the globe, the CS may add a secondary replica near users during their peak times. Once the peak time has passed, in order to reduce costs, the CS may decide to remove the added, but now lightly used, secondary replica.

Executing *add_secondary*($site_i$) is straightforward. A dedicated thread is dispatched to copy objects from the primary replica to the secondary one. Once the whole tablet is copied to the secondary replica, the new configuration becomes available to clients. Clients with the old configuration may continue submitting read operations to previously known replicas, and they eventually will become aware of the newly added secondary replica at $site_i$.

Rank	Consistency	Latency(ms)	Utility
1	RMW	50	1
2	Monotonic Read	50	0.5
3	Eventual	500	0

Figure 3: SLA of an online multiplayer game

Executing *remove_secondary*($site_i$) is also simple. The CS removes the secondary replica from the current configuration. In addition, a thread is dispatched to physically delete the secondary replica.

3.4.3 Change Primary Replica

In cases where the system maintains a single primary site, the CS may decide to change the location of the primary replica. For instance, consider the example given in Section 3.4.1. The CS may detect that adjusting the synchronization period between the primary and secondary replicas cannot improve the utility. In this case, the CS may decide to swap the primary and secondary replica roles. During peak times in India, the secondary replica in South Asia becomes the primary replica. Likewise, during peak times in Brazil, the replica in Brazil becomes primary.

The CS calls the *change_primary*($site_i$) operation to make the configuration change. If a secondary replica does not exist in $site_i$, the operation is performed in three steps. First, the CS creates a new empty replica at $site_i$. It also invalidates the configuration cached in clients. As we shall see later, when a cached configuration is invalid, a client needs to contact the CS when executing certain operations. Second, once every cached configuration becomes invalid, the CS makes $site_i$ a WRITE_ONLY primary site. In this mode, all write operations are forwarded to both the primary site and $site_i$, but $site_i$ is not allowed to execute read operations. Finally, once $site_i$ catches up with the primary replica, the CS makes it the solo primary site. If a replica exists in $site_i$, the first step is skipped. We will explain the implementation of this operation in Section 5.3.

3.4.4 Add Primary Replica

For applications that require clients to read up-to-date data as fast as possible, the system may benefit from having multiple primary sites that are strongly consistent. In multi-primary mode, write operations are applied synchronously in several sites before the client is informed that the operation has completed.

Operation	Effect	Cost
Decrease synchronization period of secondary replica at $site_i$	Increase hit ratios of subSLAs with bounded, causal, or RMW consistencies for clients near $site_i$	Increase in communication
Add $site_i$ as a secondary replica	Increase hit ratios of subSLAs with eventual or intermediary consistencies for clients near $site_i$	Additional storage; increased communication
Upgrade $site_i$ from secondary to primary, and downgrade $site_j$ from primary to secondary	Increase hit ratios of subSLAs with strong or intermediary consistency for clients near $site_i$; decrease hit ratios of subSLAs with strong or intermediary consistency for clients near $site_j$	No change
Add $site_i$ as a primary replica (upgraded from secondary)	Increase hit ratios of subSLAs with strong or intermediary consistency for clients near $site_i$	Increased communication; increased write latency

Figure 4: Summary of Common Reconfiguration Operations, Effects on Hit Ratios, and Costs.

The operation that performs the configuration transformation is called $add_primary(site_i)$. Its execution is very similar to $change_primary(site_i)$ with one exception. In the third step, instead of making the WRITE_ONLY $site_i$ the solo primary, $site_i$ is added to the list of primary replicas, thereby making the system multi-primary. In this mode, multiple rounds of operations are needed to execute a write. The protocol that we use is described in Section 5.2.3.

3.4.5 Summary

Figure 4 summarizes the reconfiguration operations that are generally considered by the CS (inverse and other less common operations are not shown). Note that the listed effects are only potential benefits. Adjusting the synchronization period or adding a secondary replica to $site_i$ does not impact the observed consistency or write latency of clients that are not near this site. These operation can possibly increase the hit ratios of subSLAs with intermediary consistencies observed by clients close to $site_i$. Adding a secondary replica can increase the hit ratios of subSLAs with eventual consistency. Making $site_i$ the solo primary increases the hit ratios of subSLAs with both strong and intermediary consistencies for clients close to $site_i$. However, clients close to the previous primary replica now may miss subSLAs with strong or intermediary consistencies. Adding a primary replica can boost strong consistency without having a negative impact on read operations; but, it increases the cost of write operations for all clients.

4 Client Execution Modes

Since the CS may reconfigure the system periodically, clients need to be aware of possible changes in the locations of primary and secondary replicas. Instead of clients asking the CS for the latest configuration before executing each operation, Tuba allows clients to cache the configuration of a tablet (called the *cview*) and use it for performing read and write operations. In this section, we explain how clients avoid two potential safety violations: (i) performing a read operation with strong consistency on a non-primary replica, or (ii) executing a write operation on a non-primary replica.

Based on the freshness of a client's cview, the client is either in fast or slow mode. Roughly speaking, a client is in the fast mode for a given tablet if it knows that it has the latest configuration. That is, it knows exactly the locations of primary and secondary replicas, and it is guaranteed that the configuration will not change in the near future. On the other hand, whenever a client suspects that a configuration may have changed, it enters slow mode until it refreshes its local cache.

Initially, every client is in slow mode. In order to enter fast mode, a client requests the latest configuration of a tablet (Figure 5). If the CS has not scheduled a change to the location of a primary replica, the client obtains the current configuration along with a promise that the CS will not modify the set of primary replicas within the next Δ seconds. Suppose the duration from when the client issues its request to when it receives the latest configuration is measured to be δ seconds. The client then enters the fast mode for $\Delta - \delta$ seconds. During this period, the client is sure that the CS will not perform a reconfiguration that compromises safety.

In order to remain in fast mode, a client needs to periodically refresh its cview. As long as it receives the latest configuration within the fast mode window,

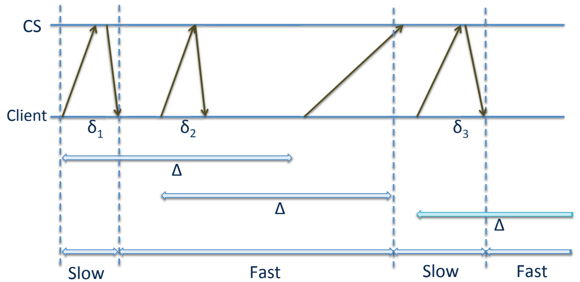


Figure 5: Clients Fast and Slow Execution Modes

it will remain in fast mode, and its fast mode window is extended.

The CS can force all clients to enter slow mode at any time by preventing them from refreshing their configuration views. This feature is used before executing *change_primary*() and *add_primary*() operations (see Section 5.3).

Fast Mode. When a client is in fast mode, read and single-primary write operations involve a single round-trip to one selected replica. No additional overhead is imposed on these operations. Multi-primary write operations use a three-phase protocol in fast or slow mode (see Section 5.2.3).

Slow Mode. Being in slow mode (for a given tablet) means that the client is not totally sure about the latest configuration, and the client needs to take some precautions. Slow mode has no affect on read operations with relaxed consistency, i.e., with any desired consistency except strong consistency. Because read operations with strong consistency must always go to a primary replica, when a client is in slow mode it needs to confirm that such an operation is indeed executed at a current primary replica. Upon completion of a strong consistency read, the client validates that the responding replica selected from its cview is still a primary replica. If not, the client retries the read operation.

Unlike read operations, write operations are more involved when a client is in slow mode. More precisely, any client in slow mode that wishes to execute a write operation on a tablet needs to take a non-exclusive lock on the tablet's configuration before issuing the write operation. On the other hand, the CS needs to take an exclusive lock on the configuration if it decides to change the set of primary replicas. This lock procedure is required to ensure the linearizability [7] of write operations.

5 Implementation

Tuba is built on top of Microsoft Azure Storage (MAS) [3] and provides a similar API for reading and writing blobs. Every MAS storage account is associated with a particular storage site. Although MAS supports Read-Access Geo-Redundant Storage (RA-GRS) in which both strong and eventual consistencies are provided, it lacks intermediary consistencies, and replication is limited to a single primary site and a single secondary site. Our implementation extends MAS with: (i) multi-site geo-replication (ii) consistency-based SLAs, and (iii) automatic reconfiguration.

A user of Tuba supplies a set of storage accounts. This set determines all available sites for replication. The CS then selects primary and secondary replica sites by choosing storage accounts from this set. Thus, a configuration is a set of MAS storage accounts tagged with PRIMARY or SECONDARY.

In the rest of this section, we explain the communication between clients and the CS, and how operations are implemented in Tuba. We ignore the implementation of consistency guarantees and consistency-based SLAs since these aspects of Tuba are taken directly from the Pileus system [15].

5.1 Communication

Clients communicate with the CS through a designated Microsoft Azure Storage container. Clients periodically write their latency and hit/miss ratios to storage blobs in this shared container. The CS reads this information and stores the latest configuration as a blob in this same container. Likewise, clients periodically read the latest configuration blob from the shared container and cache it locally.

As we explained in Section 4, when a client reads the latest configuration, it enters fast mode for $\Delta - \delta$ seconds. Since there is no direct communication between the client and the CS, we also need to ensure that the CS does not modify a primary replica and install a new configuration within the next Δ seconds. Our solution is simple. When the CS wants to perform certain reconfiguration operations (i.e., changing or adding a primary replica), it writes a special reconfiguration-in-progress (*RiP*) flag to the configuration blob's metadata. The CS then waits for at least Δ seconds before installing the new configuration. If a client fails to refresh its cview on time or if it finds that the *RiP* flag is set, then the client enters slow mode. Once the CS completes the operations needed to reconfigure the system, it overwrites the configuration blob with the latest configuration

and clears the *RiP* flag. Clients will re-enter fast mode when they next retrieve the new configuration.

5.2 Client Operations

5.2.1 Read Operation

For each read operation submitted by an application, the client library selects a replica based on the client's latency, cview, and a provided SLA (as in Pileus). The client then sends a read request to the chosen replica. Upon receiving a reply, if the client is in fast mode or if the read operation does not expect strong consistency, the data is returned immediately to the application. Otherwise, the client confirms that the contacted replica had been the primary replica at the time it answered the read request. More precisely, when a client receives a read reply message in slow mode, it reads the latest configuration and confirms that the timestamp of the configuration blob has not changed.

5.2.2 Single-primary Write Operation

To execute a single-primary write operation, a client first checks that it is in fast mode and that the remaining duration of the fast mode interval is longer than the expected time to complete the write operation. If not, it refreshes its cview. Assuming the *RiP* flag is not set, the client then writes to the primary replica. Once the client receives a positive response to this write operation, the client checks that it is still in fast mode. If so, the write operation is finished. If the write operation takes more time than expected such that the client enters slow mode during the execution of the write operation, the client confirms that the primary replica has not changed.

When a client discovers a reconfiguration in progress and remains in slow mode, we considered two approaches for performing writes. The simplest approach is for the client to wait until a new configuration becomes available. In other words, it could wait until the *RiP* flag is removed from the configuration blob's metadata. The main drawback is that no write operation is allowed on the tablet being reconfigured for Δ seconds and, during this period, the CS does nothing while waiting for all clients to enter slow mode.

Instead, Tuba allows a client in slow mode to execute a write operation by taking a lock. A client acquires a non-exclusive lock on the configuration to ensure that the CS does not change the primary replica before it executes the write operation. The CS, on the other hand, grabs an exclusive lock on the configuration before changing it. This locking mechanism is implemented as follows using MAS's existing lease support. To take a non-exclusive lock on the configuration, a client obtains a lease on the configuration blob and stores the lease-id as metadata in the blob. Other clients wishing to take a non-exclusive lock simply read the lease-id from the blob's metadata and renew the lease. To take an exclusive lock, the CS breaks the client's lease and removes the lease-id from the metadata. The CS then acquires a new lease on the configuration blob. Note that no new write is allowed after this point. After some safe threshold equal to the maximum allowed leased time, the CS updates the configuration.

5.2.3 Multi-primary Write Operation

Tuba permits configurations in which multiple servers are designated as primary replicas. A key implementation challenge was designing a protocol that atomically updates any number of replicas on conventional storage servers and that operates correctly in the face of concurrent readers and writers. Our multi-primary write protocol involves three phases: one in which a client marks his intention to write on all primary replicas, one where the client updates all of the primaries, and one where the client indicates that the write is complete. To guard against concurrent writers, we leverage the concept of ETags in Microsoft Azure, which is also part of the HTML 1.1 specification. Each blob has a string property called an ETag that is updated whenever the blob is modified. Azure allows clients to perform a conditional write operation on a blob; the write operation executes only if the provided ETag has not changed.

When an application issues a write operation to a storage blob and there are multiple primary replicas, the Tuba client library performs the following steps.

Step 1: Acquire a non-exclusive lock on the configuration blob. This step is the same as previously described for a single-primary write in slow mode. In this case, the configuration is locked even if the client is in fast mode since the multi-primary write may take longer than Δ seconds to complete. This ensures that the client knows the correct set of primary replicas throughout the protocol.

Step 2: At the main primary site, add a special write-in-progress (*WiP*) flag to the metadata of the blob being updated. The main primary site is the one listed first in the set of primary replicas. This metadata write indicates to readers that the blob is being updated, and it returns an ETag that is used later when the data is actually written. Updates to different blobs can take place in parallel.

Step 3: Write the *WiP* flag to the blob's metadata

on all other primary replicas. Note that these writes can be done in any order or in parallel.

Step 4: Perform the write on the main primary site using the ETag acquired in Step 2. Note that since writes are performed first at the main primary, this replica always holds the *truth*, i.e. the latest data. Other primary replicas hold stale data at this point. This conditional write may fail because the ETag is not current, indicating that another client is writing to the same blob. In the case of concurrent writers, the last writer to set the *WiP* flag will successfully write to the main primary replica; clients whose writes fail abandon the write protocol and possibly retry those writes later.

Step 5: Perform conditional writes on all the other primary replicas using the previously aquired Etags. These writes can be done in parallel. Again, a failed write indicates that a concurrent write is in progress. In this case, this client stops the protocol even though it may have written to some replicas already; such writes will be (or may already have been) overwritten by the latest writer (or by a recovery process as discussed in section 5.4).

Step 6: Clear the *WiP* flags in the metadata at all non-main primary sites. These flags can be cleared in any order or in parallel. This allows clients to now read from these primary replicas and obtain the newly written data. To ensure that one client does not prematurely clear the flag while another client is still writing, these metadata updates are performed as conditional writes using the ETags obtained from the writes in the previous step.

Step 7: Clear the *WiP* flag in the metadata on the main primary using a conditional write with the ETag obtained in Step 4. Because this is done as the final step, clients can check if a write is in progress simply by reading the metadata at the main primary replica.

An indication that the write has been successfully completed can be returned to the caller at any time after Step 4 where the data is written to the main primary. Waiting until the end of the protocol ensures that the write is durable since it is held at multiple primaries.

If a client attempts a strongly consistent read while another client is performing a multi-primary write, the reader may obtain a blob from the selected primary replica whose metadata contains the *WiP* flag. In this case, the client redirects its read to the main primary replica who always holds the latest data. Relaxed consistency reads, to either primary or secondary replicas, are unaffected by writes in progress.

5.3 CS Reconfiguration Operations

In this section, we only explain the implementation of *change_primary*() and *add_primary*() since the implementation details of adjusting a synchronization period and adding/removing secondary replicas are straightforward.

As we explained before, *change_primary*($site_i$) is the operation required for making $site_i$ the solo primary. If a secondary replica does not exist in $site_i$, the operation is performed in three steps. Otherwise, the first step is skipped.

Step 1: The CS starts by creating a replica at $site_i$, and synchronizing it with the primary replica.

Step 2: Before making $site_i$ the new primary replica, the CS synchronizes $site_i$ with the existing primary replica. Because write operations can run concurrently with a *change_primary*($site_i$) operation, $site_i$ might never be able to catch up with the primary replica. To address this issue, the CS first makes $site_i$ a WRITE_ONLY replica by creating a new temporary configuration. As its name suggests, write operations are applied to both WRITE_ONLY replicas and primary replicas (using the multi-primary write protocol described previously).

The CS installs this configuration as follows:

(i) It writes the *RiP* flag to the configuration blob's metadata, and waits Δ seconds to force all clients into slow mode.

(ii) Once all clients have entered the slow mode, the CS breaks the lease on the configuration blob and removes the lease-id from the metadata.

(iii) It then acquires a new lease on the blob and waits for some safe threshold.

(iv) Once the threshold is passed, the CS safely installs the temporary configuration, and removes the *RiP* flag.

Consequently, clients again switch to fast mode execution while the $site_i$ replica catches up with the primary replica.

Step 3: The final step is to make $site_i$ the primary replica, once $site_i$ is completely up-to-date. The CS follows the procedure explained in the previous step to install a new configuration where the old primary replica is downgraded to a secondary replica, and the WRITE_ONLY replica is promoted to be the new primary. Once the new configuration is installed, $site_i$ is the sole primary replica.

Note that write operations are blocked from the time when the CS takes an exclusive lease on the configuration blob until it installs the new configuration in both steps 2 and 3. However, this duration is short: a round trip latency from the CS to the configuration blob plus the safe threshold.

The *add_primary*() operation is implemented exactly like *change_primary*() with one exception. In the third step, instead of making $site_i$ the solo primary, this site is added to the list of primary replicas.

5.4 Fault-Tolerance

Replica Failure. A replica being unavailable should be a very rare occurrence since each of our replication sites is a collection of three Azure servers in independent fault domains. In any case, failed replicas can easily be removed from the system through reconfiguration. Failed secondary replicas can be ignored by clients, while failed primary replicas can be replaced using previously discussed reconfiguration operations.

Client Failure. Most read and write operations from clients are performed at a single replica and maintain no locks or leases. The failure of one client during such operations does not adversely affect others. However, Tuba does need to deal explicitly with client failures that may leave a multi-primary write partially completed. In particular, a client may crash before successfully writing to all primary replicas or before removing the *WiP* flags on one or more primary replicas.

When a client, through normal read and write operations, finds that a write to a blob has been in progress for an inordinate amount of time, it invokes a recovery process to complete the write. The recovery process knows that the main primary replica holds the truth. It reads the blob from the main primary and writes its data to the other primary replicas using the multi-write protocol described earlier. Having multiple recovery processes running simultaneously is acceptable since they all will be attempting to write the same data. The recovery process, after successfully writing to every primary replica, clears all of the *WiP* flags for the recovered blob.

CS Failure. Importantly, the Tuba design does not depend on an active CS in continuous operation. The CS may run only occasionally to check whether a reconfiguration is warranted. Since clients read the latest configuration directly from the configuration blob, and do not rely on responses from the CS, they can stay in fast mode even when the CS is not available as long as the configuration blob is available (and the *RiP* flag is not set). Since the configuration blob is replicated in MAS, it obtains the high-availability guarantees provided by Azure. If higher availability is desired, the configuration blob could be replicated across sites using Tuba's own multi-primary write protocol.

The only troubling scenario is if the CS fails while in the midst of a reconfiguration leaving the *RiP* flag set on the configuration blob. This is not a concern when the CS fails while adjusting a synchronization period or adding/removing a secondary replica. Likewise, a failure before the second step of changing/adding a primary replica does not pose any problem. Even if a CS failure leaves the *RiP* flag set, clients can still perform reads and writes in slow mode.

Recovery is easy if the CS fails during step 2 or during step 3 of changing/adding a primary replica (i.e., after setting the *RiP* flag and before clearing it). When the CS wants to performs a reconfiguration, it obtains an ETag upon setting the *RiP* flag. To install a new configuration, the CS writes the new configuration conditional on the obtained ETag.

A client clears the *RiP* flag upon waiting too long in slow mode. Doing so will prevent the CS from writing a new configuration blob and abort any reconfiguration in progress in the unlikely event that the CS had not crashed but was simply operating slowly. In other words, the CS cannot write the new configuration if some client had impatiently cleared the *RiP* flag and consequently changed the configuration blob's ETag.

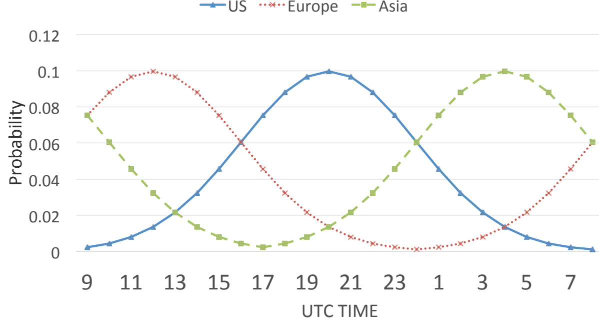
Finally, if the CS fails after step 2 of adding/changing a primary replica, clients can still enter fast mode. In case the CS was executing *change_primary*() before its crash, write operations will execute in multi-primary mode. Thus, they will be slow until the CS recovers and finishes step 3.

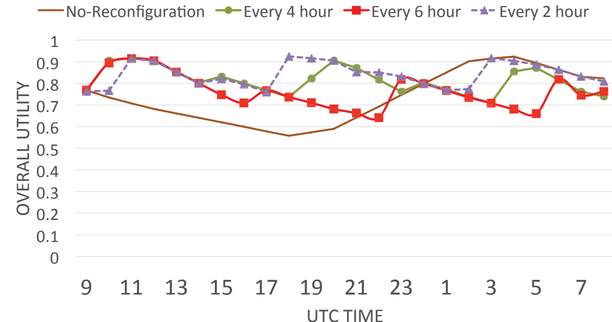
6 Evaluation

In this section, we present our evaluation results, and show how Tuba improves the overall utility of the system compared with a system that does not perform automatic reconfiguration.

6.1 Setup and Benchmark

To evaluate Tuba, we used three storage accounts located in the South US (SUS), West Europe (WEU), and South East Asia (SEA). We modeled the number of active clients with a normal distribution, and placed them in the US West Coast, West Europe, and Hong Kong (Figure 6). This is to mimic the workload of clients in different parts of the world during working hours. The mean of the normal distribution is set to 12 o'clock local time, and the variance is set to 8 hours. Considering the above





	SUS	WEU	SEA
US Clients (West US)	53	153	190
Europe Clients (West Europe)	132	<1	277
Asia Clients (Hong Kong)	204	296	36

Figure 6: Client Distribution and Latencies (in ms)

Rank	Consistency	Latency(ms)	Utility
1	Strong	100	1
2	RMW	100	0.7
3	Eventual	250	0.5

Figure 7: SLA for Evaluation

	Reconf. Every		
	6h	4h	2h
AOU	0.76	0.81	0.85
AOU Impr. over No Reconf.	5%	12%	18%
AOU Impr. over Max. Ach.	20%	45%	65%

AOU: Averaged Overall Utility in 24 hours;
No Reconf. AOU: 0.72; Max. Ach. AOU: 0.92

Figure 8: Utility improvement with different reconfiguration rates

normal distribution, the number of online clients at each hour is computed as a total number of clients times the probability distribution at that hour. The total number of clients at each site is 150 over a 24 hour period. Hence, each tablet is accessed by 450 distinct clients in one day.

We used the YCSB benchmark [6] with workload B (95% Reads and 5% writes) to generate the load. Each tablet contains 10^5 objects, and each object has a 1KB payload. Figure 7 shows the SLA used in our evaluation, which resembles one used by a social networking application [15].

The initial setup places the primary replica in SEA and a secondary replica in WEU. We set the geo-replication factor to two, allowing the CS to replicate a tablet in at most two datacenters. Moreover, we disallowed multi-primary schemes during reconfigurations.

6.2 Macroscopic View

Figure 8 compares the overall utility for read operations when reconfiguration happens every 2, 4, and 6 hours over a 24 hour period, and when no reconfiguration happens. We note that without reconfiguration Tuba performs exactly as Pileus. The average overall utility (AOU) is computed as the average utility delivered for all read operations from all clients. The average utility improvement depends on how frequently the CS performs reconfigurations. When no reconfiguration happens in the system, the AOU in the 24 hour period is 0.72. Observe that without constraints, the maximum achievable AOU would have been 1. However, limiting replication to two datacenters and a single primary decreases the maximum achievable AOU to 0.92.

Performing a reconfiguration every 6 hours improves the overall utility for almost 12 hours, and degrades it for 8 hours. This results in a 5 percent AOU improvement. When reconfiguration happens every 4 hours, the overall utility improves for 16 hours. This leads to a 12 percent AOU improvement. Finally, with 2 hour reconfigurations, AOU is improved 18 percent. Note that this improvement is 65 percent of the maximum possible improvement.

Interestingly, when no reconfiguration happens, the overall utility is better than other configurations around UTC midnight. The reason behind this phenomena is that at UTC midnight, the original replica placement is well suited for the client distribution at that time.

Figure 9 shows the hit percentages of different subSLAs. With no reconfiguration, 34% of client reads return eventually consistent data (i.e., hit the third subSLA). With 2 hour reconfigurations, Tuba

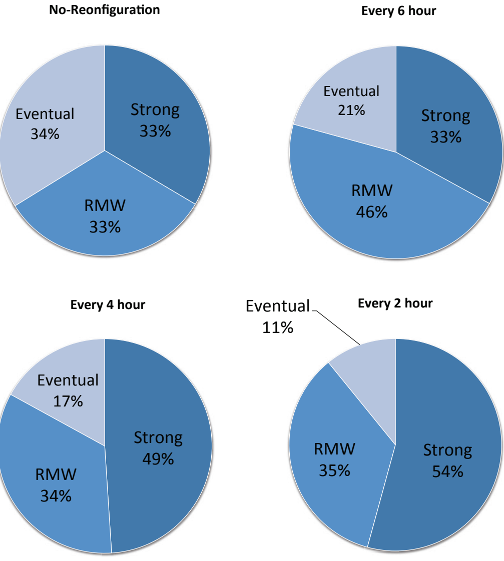
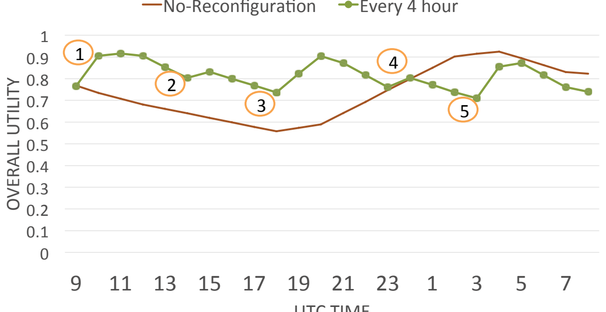


Figure 9: Hit Percentage of subSLAs



Ep och	Configuration Pri.	Sec.	Reconfiguration Operation
0	SEA	WEU	$change_primary(WEU)$
1	WEU	SEA	$add_secondary(SUS)$ $remove_secondary(SEA)$
2	WEU	SUS	$change_primary(SUS)$
3	SUS	WEU	$add_secondary(SEA)$ $remove_secondary(WEU)$
4	SUS	SEA	$change_primary(SEA)$
5	SEA	SUS	

Figure 10: Tuba with Reconfigurations Every 4 hour

reduces this to 11% (a 67% improvement). Likewise, the percentage of reads returning strongly consistent data increases by around 63%.

Although the computed AOU depends heavily on the utility values specified in the SLA, we believe that the qualitative comparisons in this study are insensitive to the specific values. Certainly, the hit percentages in Figure 9 would be unaffected by varying utilities as long as the rank order of the subSLAs is unchanged.

In addition to reduced utility, systems without support for automatic reconfiguration have additional drawbacks stemming from the way they are manually reconfigured. A system administrator must stop the system (at least for certain types of configuration changes), install the new configuration, inform clients of the new configuration, and then restart the system. Such systems are unable to perform frequent reconfigurations. Moreover, the effect of a reconfiguration on throughput can be substantial since all client activity ceases while the reconfiguration is in progress.

6.3 Microscopic View

Figure 10 shows how Tuba adapts the system configuration in our experiment where reconfiguration happens every 4 hours. The first five reconfigurations are labeled on the plot. Initially, the primary replica is located in SEA, and the secondary replica is located in WEU. Upon the first reconfiguration, the CS decides to make WEU the primary replica. Though the number of clients in Asia is decreasing at this time, the overall utility stays above 0.90 for two hours before starting to degrade.

The second reconfiguration happens around 2PM (UTC time) when the overall utility is decreased by 10%. At this time, the CS detects poor utility for users located in the US, and decides to move the secondary replica from SEA to SUS. Since the geo-replication factor is set to 2, the CS necessarily removes the secondary replica in SEA to comply with the constraint. At 6PM, the third reconfiguration happens, and SUS becomes the primary replica. This reconfiguration improves the AOU to more than 0.90. In the fourth reconfiguration, the CS decides to create a secondary replica again in the SEA region. Like the second reconfiguration, in order to respect the geo-replication constraint, the secondary replica in WEU is removed. Note that the fourth reconfiguration is suboptimal since the CS does not predict clients' future behavior and solely focuses on their past behavior. A better reconfiguration would have been to make SEA the primary replica rather than the secondary replica. After 4 hours, the CS performs another reconfiguration and again is able to boost the overall utility of the system.

Although the CS performs $adjust_sync_period()$ with two hour reconfiguration intervals, this operation is never selected by the CS when reconfigurations happen every 4 hours. This is because changing the primary or secondary replica boosts the util-

	Fast Mode		Slow Mode	
	Read	Write	Read	Write
Client in Europe	54	143	270	785
Client in Asia	297	899	533	1598

Figure 11: Average Latency (in ms) of Read/Write Operations in Fast and Slow Modes

ity enough that reducing the synchronization period would result in little additional benefit.

6.4 Fast Mode vs. Slow Mode

In this experiment, we compare the latency of read and write executions in fast and slow modes. Since the latency of read operations with any consistency other than strong does not change in fast and slow modes, we solely focus on the latency of executing read operations with strong consistency and write operations. We placed the configuration blob in the West US (WUS) datacenter, a data tablet in West Europe (WEU), and clients in Central Europe and East Asia. The latency (in ms) between the two clients and the two storage sites are as follows:

	WEU	WUS
Client in Europe	54	210
Client in Asia	296	230

Figure 11 compares the average latencies of read and write operations in slow and fast modes. Executing strongly consistent read operations in slow mode requires also reading the configuration blob to ensure that the primary replica has not changed. Therefore, the latency of a read operation in slow mode is more than 200 ms longer than in fast mode.

Executing write operations in slow mode requires three additional RPC calls to the US (where the configuration blob is stored) in the case where no client has written a lease-id to the configuration's metadata (as in this experiment). Specifically, slow mode writes involve reading the latest configuration, taking a non-exclusive lease on the configuration blob, and writing the lease-id to the configuration's metadata. If a lease-id is already set in the configuration's metadata, the last phase is not needed, and two RPC calls are enough. We note that, with additional support from the storage servers, the overhead of write operations in slow mode could be trimmed to only one additional RPC call. This is achievable by taking or renewing the lease in one RPC call to the server that stores the configuration.

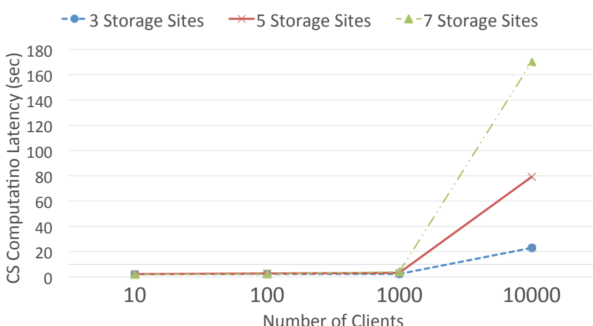


Figure 12: Scalability of the CS

6.5 Scalability of the CS

As we explained in Section 3.3, the CS considers a potentially large number of candidates when selecting a new configuration. To better understand the limitations of the selection algorithm used by our CS, we studied its scalability in practice. We put clients at four sites: East US, West US, West Europe, and Southeast Asia. Each client's SLA has three subSLAs, and all SLAs are distinct; thus, no ratio aggregation is possible. Initially, the East US site is chosen as the primary replica, and no secondary replica is deployed. We also impose the following three constraints: (i) Do not replicate in East US, (ii) Replicate in at least two sites, and (iii) Replicate in a maximum of three sites. We ran the CS on a dual-core 2.20 GHz machine with 3.5GB of memory.

Figure 12 plots the latency of computing a new configuration with 3, 5, and 7 available storage sites when the CS performs an exhaustive search of all possible configurations. With one hundred clients, it takes less than 3 seconds to compute the expected utility gain for every configuration and to select the best one. With one thousand clients, the computation time for 3 available storage sites is still less than 3 seconds, while it reaches 3.8 seconds for 7 sites. When the number of clients reaches ten thousand, the CS computes a new configuration for 3 available storage sites in 20 seconds, and for 7 available storage sites in 170 seconds.

This performance is acceptable for many systems since typically the set of cloud storage sites (i.e., the datacenters in which data can be stored) is small and reconfigurations are infrequent. For systems with very large numbers of clients and a large list of possible storage sites, heuristics for pruning the search space could yield substantial improvements and other techniques like ILP or constraint programming should be explored.

7 Related Work

Lots of previous work has focused on data placement and adaptive replication algorithms in LAN environments (e.g., [2, 8, 11–13, 18]). These techniques are not applicable for WAN environments mainly because: (i) intra-datacenter transfer costs are negligible compared to inter-datacenter costs, (ii) data should be placed in the datacenters that are closest to users, and (iii) the system should react to users' mobility around the globe. Therefore, in the remaining of this section, we only review solutions tailored specifically for WAN environments.

Kadambi et al. [9] introduce a mechanism for selectively replicating large databases globally. Their main goal is to minimize the bandwidth required to send updates and the bandwidth required to forward reads to remote datacenters while respecting policy constraints. They extend Yahoo! PNUTs [5] with a per-record selective replication policy. Their dynamic placement algorithm is based on work by Wolfson et al. [18] and responds to changes in access patterns by creating and removing replicas. They replicate all records in all locations either as a full copy or as a stub. The full replica is a normal copy of the data while the stub contains only the primary-key and some metadata. Instead of recording access patterns as in Tuba, they rely on a simple approach: a stub replica becomes full when a read operation is delivered at its location, and a full replica demotes when a write operation is observed in another location or if there has not been any read at that location for some period. Unlike Tuba, changing the primary replica is not studied in this work. Moreover, once data is inserted into a tablet, policy constraints cannot be changed. In contrast, Tuba allows modifying or adding new constraints, and the current set of constraints will be respected in the next reconfiguration cycle.

Tran et al. [16] introduce a key-value store called Nomad that allows migrating of data between datacenters. They propose and implement an abstraction called overlays. These overlays are responsible for caching and migrating object containers across datacenters. Nomad considers the following three migration policies: (i) count, (ii) time, and (iii) rate. Users can specify the number of times, a certain period, and the rate that data is accessed from the same remote location. In comparison, Tuba focuses on maximizing the overall utility of the storage system and respecting replication constraints.

Volley [1] relies on access logs to determine data locations. Their goal is to improve datacenter capacity skew, inter-datacenter traffic, and client latency. In each round, Volley computes the data placement for *all* data items, while the granularity in Tuba is a tablet. Unlike Tuba, Volley does not take into account the configuration costs or constraints. Moreover, the Volley paper does not suggest any migration mechanisms.

Venkataramani et al. [17] propose a bandwidth-constrained placement algorithm for WAN environments. Their main goal is to place copies of objects at a collection of caches to minimize access time. However, complex coordination between distributed nodes and the assumption of a fixed size for all objects makes this scheme less practical than the techniques presented in this paper.

8 Conclusion

Tuba is a replicated key-value store that, like Pileus, allows applications to specify their desired consistency and dynamically selects replicas in order to maximize the utility delivered to read operations. Additionally, Tuba automatically reconfigures itself while respecting user defined constraints so that it adapts to changes in users locations or request rates. The system is built on Microsoft Azure Storage (MAS), and extends MAS with broad consistency choices, consistency-based SLAs, and explicit geo-replication configurations.

Our experiments with clients distributed in different datacenters around the world show that Tuba with two hour reconfiguration intervals increases the reads that return strongly consistent data by 63% and improves average utility up to 18%. This confirms that automatic reconfiguration can yield substantial benefits which are realizable in practice.

Acknowledgements

We thank Marcos K. Aguilera, Mahesh Balakrishnan, and Ramakrishna Kotla for their insightful discussions as well as for their contributions to the design and implementation of the original Pileus system. We would like to also thank Pierpaolo Cincilla, Tyler Crain, Gilles Muller, Marc Shapiro, Pierre Sutra, Marek Zawirski, the anonymous reviewers, and our shepherd, Emin Gün Sirer, for their thoughtful suggestions and feedback on this work.

References

[1] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: automated data placement for geo-distributed cloud services. In *Networked Sys. Design and Implem. (NSDI)*, page 2. USENIX Association, Apr. 2010.

[2] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Usenix Annual Tech. Conf. (Usenix-ATC)*. USENIX Association, June 2000.

[3] B. Calder, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, J. Wang, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, L. Rigas, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, and J. Wu. Windows Azure Storage. In *Symp. on Op. Sys. Principles (SOSP)*, pages 143—-157, New York, New York, USA, Oct. 2011. ACM Press.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *Trans. on Computer Sys.*, 26(2):1–26, June 2008.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Symp. on Cloud Computing (SoCC)*, pages 143—-154, New York, NY, USA, 2010. ACM.

[7] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463—-492, 1990.

[8] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, OSDI '99, pages 187–200. USENIX Association, Feb. 1999.

[9] S. Kadambi, J. Chen, B. F. Cooper, D. Lomax, A. Silberstein, E. Tam, and H. Garcia-molina. Where in the World is My Data ? In *Int. Conf. on Very Large Data Bases (VLDB)*, pages 1040–1050, 2011.

[10] G. M. V. Lili Qiu, Venkata N. Padmanabhan. On the placement of web server replicas. In *Int. Conf. on Computer Communications (INFOCOM)*, pages 1587—-1596, 2001.

[11] R. R. Madhukar R. Korupolu, C. Greg Plaxton. Placement Algorithms for Hierarchical Cooperative Caching. In *Symp. on Discrete Algorithms (SODA)*, pages 586–595. Society for Industrial and Applied Mathematics, 1999.

[12] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. In *Euro. Conf. on Comp. Sys. (EuroSys)*, number 4, page 89, New York, New York, USA, Oct. 2006. ACM.

[13] C. Stewart, S. Dwarkadas, and M. Scott. *Distributed Systems Online*, 05(10):1–1, Oct. 2004.

[14] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149. IEEE Computer Society, Sept. 1994.

[15] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Symp. on Op. Sys. Principles (SOSP)*, pages 309–324, New York, New York, USA, Nov. 2013. ACM Press.

[16] N. Tran, M. K. Aguilera, and M. Balakrishnan. Online migration for geo-distributed storage systems. In *Usenix Annual Tech. Conf. (Usenix-ATC)*, Berkeley, CA, USA, 2011. USENIX Association.

[17] A. Venkataramani, P. Weidmann, and M. Dahlin. Bandwidth constrained placement in a WAN. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 134–143, New York, New York, USA, Aug. 2001. ACM Press.

[18] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *Trans. on Database Sys.*, 22(2):255–314, June 1997.

f4: Facebook's Warm BLOB Storage System

Subramanian Muralidhar*, Wyatt Lloyd†*, Sabyasachi Roy*, Cory Hill*, Ernest Lin*, Weiwen Liu*, Satadru Pan*, Shiva Shankar*, Viswanath Sivakumar*, Linpeng Tang‡*, Sanjeev Kumar*

*Facebook Inc., †University of Southern California, ‡Princeton University

Abstract

Facebook's corpus of photos, videos, and other Binary Large OBjects (BLOBs) that need to be reliably stored and quickly accessible is massive and continues to grow. As the footprint of BLOBs increases, storing them in our traditional storage system, Haystack, is becoming increasingly inefficient. To increase our storage efficiency, measured in the effective-replication-factor of BLOBs, we examine the underlying access patterns of BLOBs and identify temperature zones that include hot BLOBs that are accessed frequently and warm BLOBs that are accessed far less often. Our overall BLOB storage system is designed to isolate warm BLOBs and enable us to use a specialized warm BLOB storage system, f4. f4 is a new system that lowers the effective-replication-factor of warm BLOBs while remaining fault tolerant and able to support the lower throughput demands.

f4 currently stores over 65PBs of logical BLOBs and reduces their effective-replication-factor from 3.6 to either 2.8 or 2.1. f4 provides low latency; is resilient to disk, host, rack, and datacenter failures; and provides sufficient throughput for warm BLOBs.

1. Introduction

As Facebook has grown, and the amount of data shared per user has grown, storing data efficiently has become increasingly important. An important class of data that Facebook stores is Binary Large OBjects (BLOBs), which are immutable binary data. BLOBs are created once, read many times, never modified, and sometimes deleted. BLOB types at Facebook include photos, videos, documents, traces, heap dumps, and source code. The storage footprint of BLOBs is large. As of February 2014, Facebook stored over 400 billion photos.

Haystack [5], Facebook's original BLOB storage system, has been in production for over seven years and is designed for IO-bound workloads. It reduces the number of disk seeks to read a BLOB to almost always one and triple replicates data for fault tolerance and to support a high request rate. However, as Facebook has grown and evolved, the BLOB storage workload has changed. The types of BLOBs stored have increased. The diversity in size and create, read, and delete rates has increased. And, most importantly, there is now a large and increasing number of BLOBs with low request rates. For these BLOBs, triple replication results in over provisioning from a throughput perspective. Yet, triple replication also provided important fault tolerance guarantees.

Our newer f4 BLOB storage system provides the same fault tolerance guarantees as Haystack but at a lower effective-replication-factor. f4 is simple, modular, scalable, and fault tolerant; it handles the request rate of BLOBs we store it in; it responds to requests with sufficiently low latency; it is tolerant to disk, host, rack and datacenter failures; and it provides all of this at a low effective-replication-factor.

We describe f4 as a *warm* BLOB storage system because the request rate for its content is lower than that for content in Haystack and thus is not as "hot." Warm is also in contrast with cold storage systems [20, 40] that reliably store data but may take days or hours to retrieve it, which is unacceptably long for user-facing requests. We also describe BLOBs using temperature, with hot BLOBs receiving many requests and warm BLOBs receiving few.

There is a strong correlation between the age of a BLOB and its temperature, as we will demonstrate. Newly created BLOBs are requested at a far higher rate than older BLOBs. For instance, the request rate for week-old BLOBs is an order of magnitude lower than for less-than-a-day old content for eight of nine examined types. In addition, there is a strong correlation between age and the deletion rate. We use these findings to inform our design: the lower request rate of warm BLOBs enables us to provision a lower maximum throughput for f4 than Haystack, and the low delete rate for warm BLOBs enables us to simplify f4 by not needing to physically reclaim space quickly after deletes. We also use our finding to identify warm content using the correlation between age and temperature.

Facebook's overall BLOB storage architecture is designed to enable warm storage. It includes a caching stack that significantly reduces the load on the storage systems and enables them to be provisioned for fewer requests per BLOB; a transformer tier that handles computational-intense BLOB transformation and can be scaled independently of storage; a router tier that abstracts away the underlying storage systems and enables seamless migration between them; and the hot storage system, Haystack, that aggregates newly created BLOBs into volumes and stores them until their request and delete rates have cooled off enough to be migrated to f4.

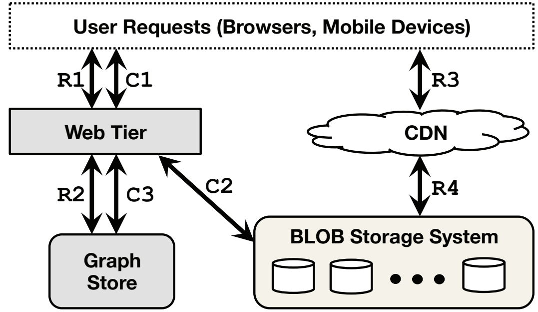


Figure 1: Reading (R1-R4) and creating (C1-C3) BLOBs.

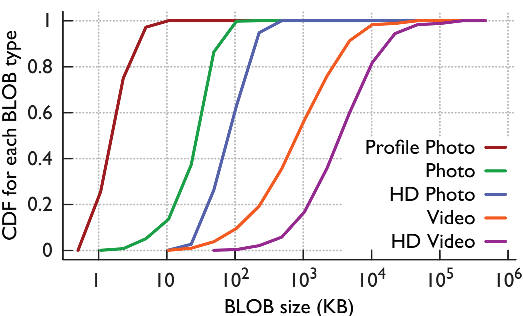


Figure 2: Size distribution for five BLOB types.

f4 stores volumes of warm BLOBs in cells that use distributed erasure coding, which uses fewer physical bytes than triple replication. It uses Reed-Solomon(10,4) [46] coding and lays blocks out on different racks to ensure resilience to disk, machine, and rack failures within a single datacenter. Is uses XOR coding in the wide-area to ensure resilience to datacenter failures. f4 has been running in production at Facebook for over 19 months. f4 currently stores over 65PB of logical data and saves over 53PB of storage.

Our contributions in this paper include:

- A case for warm storage that informs future research on it and justifies our efforts.
- The design of our overall BLOB storage architecture that enables warm storage.
- The design of f4, a simple, efficient, and fault tolerant warm storage solution that reduces our effective-replication-factor from 3.6 to 2.8 and then to 2.1.
- A production evaluation of f4.

The paper continues with background in Section 2. Section 3 presents the case for warm storage. Section 4 presents the design of our overall BLOB storage architecture that enables warm storage. f4 is described in Section 5. Section 6 covers a production evaluation of f4, Section 7 covers lessons learned, Section 8 covers related work, and Section 9 concludes.

2. Background

This section explains where BLOB storage fits in the full architecture of Facebook. It also describes the different types of BLOBs we store and their size distributions.

2.1 Where BLOB Storage Fits

Figure 1 shows how BLOB storage fits into the overall architecture at Facebook. BLOB creates—e.g., a video upload—originate on the web tier (C1). The web tier writes the data to the BLOB storage system (C2) and then stores the handle for that data into our graph store (C3), Tao [9]. The handle can be used to retrieve or delete the BLOB. Tao associates the handle with other elements of the graph, e.g., the owner of a video.

BLOB reads—e.g., watching a video—also originate on the web tier (R1). The web tier accesses the Graph Store (R2) to find the necessary handles and constructs a URL that can be used to fetch the BLOB. When the browser later sends a request for the BLOB (R3), the request first goes to a content distribution network (CDN) [2, 34] that caches commonly accessed BLOBs. If the CDN does not have the requested BLOB, it sends a request to the BLOB storage system (R4), caches the BLOB, and returns it to the user. The CDN shields the storage system from a significant number of requests on frequently accessed data, and we return to its importance in Sections 4.1.

2.2 BLOBs Explained

BLOBs are immutable binary data. They are created once, read potentially many times, and can only be deleted, not modified. This covers many types of content at Facebook. Most BLOB types are user facing, such as photos, videos, and documents. Other BLOB types are internal, such as traces, heap dumps, and source code. User-facing BLOBs are more prevalent so we focus on them for the remainder of the paper and refer to them as simply BLOBs.

Figure 2 shows the distribution of sizes for five types of BLOBs. There is a significant amount of diversity in the sizes of BLOBs, which has implications for our design as discussed in Section 5.6.

3. The Case for Warm Storage

This section motivates the creation of a warm storage system at Facebook. It demonstrates that temperature zones exist, age is a good proxy for temperature, and that warm content is large and growing.

Methodology The data presented in this section is derived from a two-week trace, benchmarks of existing systems, and daily snapshots of summary statistics. The trace includes a random 0.1% of reads, 10% of creates, and 10% of deletes.

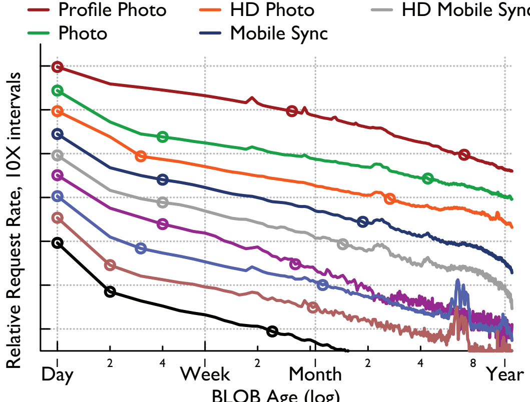


Figure 3: Relative request rates by age. Each line is relative to only itself, absolute values have been denormalized to increase readability, and points mark an order-of-magnitude decrease in request rate.

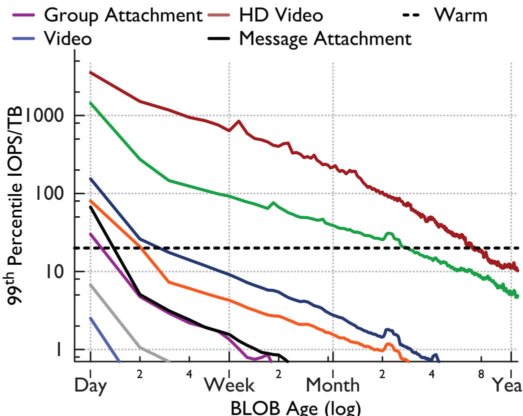


Figure 4: 99th percentile load in IOPS/TB of data for different BLOB types for BLOBs of various ages.

Data is presented for nine user-facing BLOB types. We exclude some data types from some analysis due to incomplete logging information.

The nine BLOB types include Profile Photos, Photos, HD Photos, Mobile Sync Photos [17], HD Mobile Sync Photos, Group Attachments [16], Videos, HD Videos, and Message (chat) Attachments. Group Attachments and Message Attachments are opaque BLOBS to our storage system, they can be text, pdfs, presentation, etc.

Temperature Zones Exist To make the case for warm storage we first show that temperature zones exist, i.e., that content begins as hot, receiving many requests, and then cools over time, receiving fewer and fewer requests.

Figure 3 shows the relative request rate, requests-per-object-per-hour, for content of a given age. The two-week trace of 0.1% of reads was used to create this figure. The age of each object being read is recorded and these are bucketed into 1-day intervals. We then count the number of requests to the daily buckets for each hour in the trace and report the mean—the medians are similar but noisier. Absolute values are denormalized to increase readability so each line is relative to only itself. Points mark order-of-magnitude decreases.

The existence of temperature zones is clear in the trend of decreasing request rates over time. For all nine types, content less than one day old receives more than 100 times the request rate of one-year-old content. For eight of the types the request rate drops by an order of magnitude in less than a week, and for six of the types the request rate drops by 100x in less than 60 days.

Differentiating Temperature Zones Given that temperature zones exist, the next questions to answer are how to differentiate warm from hot content and when it is safe to move content to warm storage. We define the *warm temperature zone* to include unchanging content with a low request rate. BLOBs are not modified, so the only changes are the deletes. Thus, differentiating warm from hot content depends on the request and delete rates.

First, we examine the request rate. To determine where to draw the line between hot and warm storage we consider near-worst-case request rates because our internal service level objects require low near-worst-case latency during our busiest periods of the day.

Figure 4 shows the 99th percentile or near-worst-case request load for BLOBs of various types grouped by age. The two-week trace of 0.1% of reads was used to create this figure. The age of each object read is recorded and these are bucketed into intervals equivalent to the time needed to create 1 TB of that BLOB type. For instance, if 1 TB of a type is created every 3600 seconds, then the first bucket is for ages of 0-3599 seconds, the second is for 3600-7200 seconds, and so on.[1] We then compensate for the 0.1% sampling rate by looking at windows of 1000 seconds. We report the 99th percentile request rate for these windows, i.e., we report the 99th percentile count of requests in a 1000 second window across our two-week trace for each age bucket. The 4TB disks used in f4 can deliver a maximum of 80 Input/Output Operations Per Second (IOPS) while keeping per-request latency acceptably low. The figure shows this peak warm storage throughput at 20 IOPS/TB.

For seven of the nine types the near-worst-case throughput is below the capacity of the warm storage system in less than a week. For Photos, it takes ~3 months to drop below the capacity of warm storage and for Profile Photos it takes a year.

We also examined, but did not rigorously quantify, the deletion rate of BLOB types over time. The general trend

[1] We spread newly created BLOBs over many hosts and disks, so no host or disk in our system is subject to the extreme loads on far left of Figure 4. We elaborate on this point further in Section 5.6

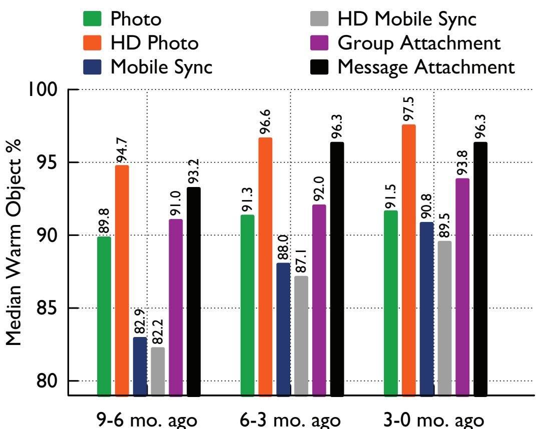


Figure 5: Median percentage of each type that was warm 9-6 months ago, 6-3 months ago, 3 months ago to now. The remaining percentage of each type is hot.

is that most deletes are for young BLOBs and that once the request rate for a BLOB drops below the threshold of warm storage, the delete rate is low as well.

Combining the deletion analysis with the request rate analysis yields an age of a month as a safe delimiter between hot and warm content for all but two BLOB types. One type, Profile Photos, is not moved to warm storage. The other, Photos, uses a three months threshold.

Warm Content is Large and Growing We finish the case for warm storage by demonstrating that the percentage of content that is warm is large and continuing to grow. Figure 5 gives the percentage of content that is warm for three-month intervals for six BLOB types.

We use the above analysis to determine the warm cutoff for each type, i.e., one month for most types. This figure reports the median percentage of content for each type that is warm in three-month intervals from 9-6 months ago, 6-3 months ago, and 3 months ago to now.

The figure shows that warm content is a large percentage of all objects: in the oldest interval more than 80% of objects are warm for all types. It also shows that the warm fraction is increasing: in the most recent interval more than 89% of objects are warm for all types.

This section showed that temperature zones exist, that the line between hot and warm content can safely be drawn for existing types at Facebook at one month for most types, and that warm content is a large and growing percentage of overall BLOB content. Next, we describe how Facebook's overall BLOB storage architecture enables warm storage.

4. BLOB Storage Design

Our BLOB storage design is guided by the principle of keeping components simple, focused, and well-matched to their job. In this section we explain volumes, describe

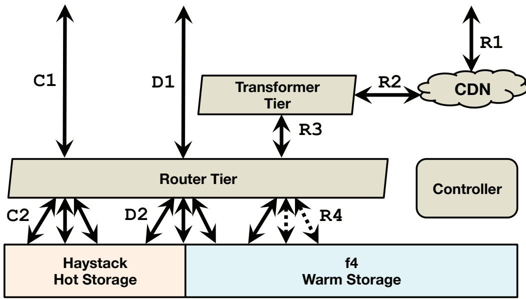


Figure 6: Overall BLOB Storage Architecture with creates (`C1-C2`), deletes (`D1-D2`), and reads (`R1-R4`). Creates are handled by Haystack, most deletes are handled by Haystack, reads are handled by either Haystack or f4.

the full design of our BLOB storage system, and explain how it enables focused and simple warm storage with f4.

Volumes We aggregate BLOBs together into logical volumes. Volumes aggregate filesystem metadata, allowing our storage systems to waste few IOPS as we discuss further below. We categorize logical volumes into two classes. Volumes are initially *unlocked* and support reads, creates (appends), and deletes. Once volumes are full, at around 100GB in size, they transition to being *locked* and no longer allow creates. Locked volumes only allow reads and deletes.

Each volume is comprised of three files: a data file, an index file, and a journal file. The data file and index files are the same as the published version of Haystack [5], while the journal file is new. The *data* file holds each BLOB along with associated metadata such as the key, the size, and checksum. The *index* file is a snapshot of the in-memory lookup structure of the storage machines. Its main purpose is allowing rebooted machines to quickly reconstruct their in-memory indexes. The *journal* file tracks BLOBs that have been deleted; whereas in the original version of Haystack, deletes were handled by updating the data and index files directly. For locked volumes, the data and index files are read-only, while the journal file is read-write. For unlocked volumes, all three files are read-write.

4.1 Overall Storage System

The full BLOB storage architecture is shown in Figure 6. Creates enter the system at the router tier (`C1`) and are directed to the appropriate host in the hot storage system (`C2`). Deletes enter the system at the router tier (`D1`) and are directed to the appropriate hosts in appropriate storage system (`D2`). Reads enter the system at the caching stack (`R1`) and, if not satisfied there, traverse through the transformer tier (`R2`) to the router tier (`R3`) that directs them to the appropriate host in the appropriate storage system (`R4`).

Controller The controller ensures the smooth functioning of the overall system. It helps with provisioning new store machines, maintaining a pool of unlocked volumes, ensuring that all logical volumes have enough physical volumes backing them, creating new physical volumes if necessary, and performing periodic maintenance tasks such as compaction and garbage collection.

Router Tier The router tier is the interface of BLOB storage; it hides the implementation of storage and enables the addition of new subsystems like f4. Its clients, the web tier or caching stack, send operations on logical BLOBs to it.

Router tier machines are identical, they execute the same logic and all have soft state copies of the logical-volume-to-physical-volume mapping that is canonically stored in a separate database (not pictured). The router tier scales by adding more machines and its size is independent of the other parts of the overall system.

For reads, a router extracts the logical volume id from the BLOB id and finds the physical mapping of that volume. It chooses one of available physical volumes—typically, the volume on the closest machine—and sends the request to it. In case of failure, a timeout fires and the request is directed to the next physical volume.

For creates, the router picks a logical volume with available space, and sends the BLOB out to all physical volumes for that logical volume. In case of any errors, any partially written data is ignored to be garbage collected later, and a new logical volume is picked for the create.

For deletes, the router issues deletes to all physical replicas of a BLOB. Responses are handled asynchronously and the delete is continually retried until the BLOB is fully deleted in case of failure.

The router tier enables warm storage by hiding the storage implementation from its clients. When a volume is migrated from the hot storage system to the warm storage system it temporarily resides in both while the canonical mapping is updated and then client operations are transparently directed to the new storage system.

Transformer Tier The transformer tier handles a set of transformations on the retrieved BLOB. For example, these transformations include resizing and cropping photos. In Facebook's older system, these computational intensive transformations were performed on the storage machines.

The transformer tier enables warm storage by freeing the storage system to focus solely on providing storage. Separating computation into its own tier allows us to scale out the storage tier and the transformer tier independently. In turn, that allows us to match the size of the storage tiers precisely to our needs. Furthermore, it enables us to choose more optimal hardware for each of these tasks. In particular, storage nodes can be designed to hold a large number of disks with only a single CPU and relatively little RAM.

Caching Stack BLOB reads are initially directed to the caching stack [2, 34] and if a BLOB is resident in one of the caches it is returned directly, avoiding a read in the storage system. This absorbs reads for popular BLOBs and decreases the request rate at the storage system. The caching stack enables warm storage by lowering its request rate.

Hot Storage with Haystack Facebook's hot storage system, Haystack, is designed to use only fully-utilized IOPS. It enables warm storage by handling all BLOB creates, handling most of the deletes, and handling a higher read rate.

Haystack is designed to fully utilize disk IOPS by:

- **Grouping BLOBs**: It creates only a small number (~100) of files with BLOBs laid out sequentially in those files. The result is a simple BLOB storage system that uses a small number of files, and bypasses the underlying file system for most metadata access.
- **Compact metadata management**: It identifies the minimal set of metadata that is needed to locate each BLOB and carefully lays out this metadata so that it fits in the available memory on the machine. This allows the system to waste very few IOPS for metadata fetches.

BLOBs are grouped into *logical volumes*. For fault tolerance and performance, each logical volume maps into multiple *physical volumes* or replicas on different hosts across different geographical regions: all physical volumes for a logical volume store the same set of BLOBs. Each physical volume lives entirely on one Haystack host. There are typically 3 physical volumes for each logical volume. Each volume holds up to millions of immutable BLOBs, and can grow to ~100GB in size.

When a host receives a read it looks up the relevant metadata—the offset in the data file, the size of the data record, and whether it has been deleted—in the in-memory hash table. It then performs a single I/O request to the data file to read the entire data record.

When a host receives a create it synchronously appends a record to its physical volume, updates the in-memory hash tables, and synchronously updates the index and journal files.

When a host receives a delete it updates the its in-memory hash tables and the journal file. The contents of the BLOB still exist in the data file. Periodically we *compact* volumes, which completely deletes the BLOB and reclaims its space.

Fault tolerance Haystack has fault tolerance to disk, host, rack, and datacenter failure through triple replication of data files and hardware RAID-6 (1.2X replication).

Two replicas of each volume are in a primary datacenter but on different racks, and thus hosts and disks. This provides resilience to disk, host, and rack failure. RAID-6 provides additional protection against disk failure. The third replica is in another datacenter and provides resilience to datacenter failure.

This scheme provides good fault tolerance and high throughput for BLOBs, but at an effective-replication-factor of $3 * 1.2 = 3.6$. This is the main limitation of Haystack: it is optimized for IOPS but not storage efficiency. As the case for warm storage demonstrated, this results in significant over replication of many BLOBs.

Expiry-Driven Content Some BLOB types have expiration times for their content. For instance, uploaded videos are stored in their original format temporary while they are transcoded to our storage formats. We avoid ever moving this expiry-driven content to f4 and keep it in Haystack. The hot storage system copes with the high delete rate by running compaction frequently to reclaim the now available space.

5. f4 Design

This section describes our design goals for warm storage and then describes f4, our warm storage system.

5.1 Design Goals

At a high level, we want our warm storage system to provide storage efficiency and to provide fault tolerance so we do not lose data or appear unavailable to our users.

Storage Efficiency One of the key goals of our new system is to improve storage efficiency, i.e., reduce the effective-replication-factor while still maintaining a high degree of reliability and performance.

The *effective replication factor* describes the ratio of actual physical size of the data to the logical size stored. In a system that maintains 3 replicas, and uses RAID-6 encoding on each node with 12 disks, the effective replication factor is 3.6.

Fault Tolerance Another important goal for our storage system is fault tolerance to a hierarchy of faults to ensure we do not lose data and that storage is always available for client requests. We explicitly consider four types of failures:

1. Drive failures, at a low single digit annual rate.
2. Host failures, periodically.
3. Rack failures, multiple time per year.
4. Datacenter failures, extremely rare and usually transient, but potentially more disastrous.

5.2 f4 Overview

f4 is our storage subsystem for warm data. It is comprised of a number of *cells*, where each cell lives entirely within one datacenter and is comprised of homogeneous hardware. Current cells use 14 racks of 15 hosts [42] with 30 4TB drives per host. We treat a cell as a unit of acquisition and as a unit of deployment and roll out.

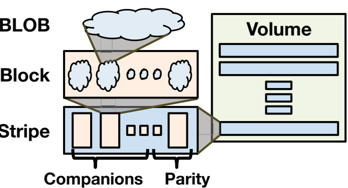


Figure 7: BLOBs in Blocks in Stripes in Volumes.

A cell is responsible for reliably storing a set of locked volumes and uses Reed-Solomon coding to store these volumes with lower storage overhead. Distributed erasure coding achieves reliability at lower-storage overheads than replication, with the tradeoff of increased rebuild and recovery times under failure and lower maximum read throughput. *Reed-Solomon coding* [46] is one of the most popular erasure coding techniques, and has been employed in a number of different systems. A Reed-Solomon(n, k) code encodes n bits of data with k extra bits of parity, and can tolerate k failures, at an overall storage size of $n + k$. This scheme protects against disk, host, and rack failures.

We use a separate XOR coding scheme to tolerate datacenter or geographic region failure. We pair each volume/stripe/block with a *buddy* volume/stripe/block in a different geographic region. We store an XOR of the buddies in a third region. This scheme protects against failure of one of the three regions. We discuss fault tolerance in Section 5.5

5.3 Individual f4 Cell

Individual f4 cells are resilient to disk, host, and rack failures and are the primary location and interface for the BLOBs they store. Each f4 cell handles only locked volumes, i.e., it only needs to support read and delete operations against that volume. The data and index files are read-only. The haystack journal files that track deletes are not present in f4. Instead, all BLOBs are encrypted with keys that are stored in an external database. Deleting the encryption key for a BLOB in f4 logically deletes it by making it unreadable.

The index files use triple replication within a cell. The files are small enough that the storage gain from encoding them is too small to be worth the added complexity.

The data file with the actual BLOB data is encoded and stored via a Reed-Solomon(n, k) code. Recent f4 cells use $n = 10$ and $k = 4$. The file is logically divided up into contiguous sequences of n blocks, each of size b. For each such sequence of n blocks, k parity blocks are generated, thus forming a logical *stripe* of size $n + k$

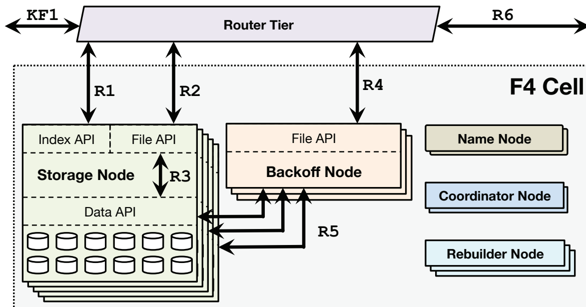


Figure 8: f4 Single Cell Architecture. `R1`–`R3` shows a normal-case read. `R1`, `R4`, `R5` shows a failure-case read. `KF1` show the encryption key fetch that happens in parallel with the rest of the read path in f4.

blocks. For a given block in a stripe, the other blocks in the stripe are considered to be its *companion* blocks. If the file is not an integral multiple of n blocks, it is zero-padded to the next multiple. In normal operation BLOBs are read directly from their data block. If a block is unavailable it can be recovered by decoding any n of its companion and parity blocks. A subset of a block, corresponding to a BLOB, can also be decoded from only the equivalent subsets of any n of its companion and parity blocks. Figure 7 shows the relationship between BLOBs, blocks, strips, and volumes.

The block-size for encoding is chosen to be a large value—typically 1 GB—for two reasons. First, it decreases the number of BLOBs that span multiple blocks and thus require multiple I/O operations to read. Second, it reduces the amount of per-block metadata that f4 needs to maintain. We avoid a larger block size because of the larger overhead for rebuilding blocks it would incur.

Figure 8 shows a f4 cell. Its components include storage nodes, name nodes, backoff nodes, rebuilder nodes, and coordinator nodes.

Name Node The name node maintains the mapping between data blocks and parity blocks and the storage nodes that hold the actual blocks. The mapping is distributed to storage nodes via standard techniques [3, 18]. Name nodes are made fault tolerant with a standard primary-backup setup.

Storage Nodes The storage nodes are the main component of a cell and handle all normal-case reads and deletes. Storage nodes expose two APIs: an *Index API* that provides existence and location information for volumes, and a *File API* that provides access to data.

Each node is responsible for the existence and location information of a subset of the volumes in a cell and exposes this through its Index API.[2] It stores the index—BLOB to data file, offset, and length—file on disk and loads them into custom data structures in memory. It also loads the location-map for each volume that maps offsets in data files to the physically-stored data blocks. Index files and location maps are pinned in memory to avoid disk seeks.

Each BLOB in f4 is encrypted with a per-BLOB encryption key. Deletes are handled outside of f4 by deleting a BLOB's encryption key that is stored in a separate key store, typically a database. This renders the BLOB unreadable and effectively deletes it without requiring the use of compaction in f4. It also enables f4 to eliminate the journal file that Haystack uses to track key presence and deletion information.

Reads (`R1`) are handled by validating that the BLOB exists and then redirecting the caller to the storage node with the data block that contains the specified BLOB.

The Data API provides data access to the data and parity blocks the node stores. Normal-case reads are redirected to the appropriate storage node (`R2`) that then reads the BLOB directly from its enclosing data block (`R3`). Failure-case reads use the Data API to read companion and parity blocks needed to reconstruct the BLOB on a backoff node.

The router tier fetches the per-BLOB encryption key in parallel with the rest of the read path, i.e., `R1`–`R3` or `R1`, `R4`, `R5`. The BLOB is then decrypted on the router tier. Decryption is computationally expensive and performing it on the router tier allows f4 to focus on efficient storage and allows decryption to be scaled independently from storage.

Backoff Nodes When there are failures in a cell, some data blocks will become unavailable, and serving reads for the BLOBs it holds will require *online reconstruction* of them from companion data blocks and parity blocks. Backoff nodes are storage-less, CPU-heavy nodes that handle the online reconstruction of request BLOBs.

Each backoff node exposes a *File API* that receives reads from the router tier after a normal-case read fails (`R4`). The read request has already been mapped to a data file, offset, and length by a primary volume-server. The backoff volume-server sends reads of that length from the equivalent offsets from all $n-1$ companion blocks and k parity blocks for the unavailable block (`R5`). Once it receives n responses it decodes them to reconstruct the requested BLOB.

This online reconstruction rebuilds only the requested BLOB, it does not rebuild the full block. Because the size of a BLOB is typically much smaller than the block

[2] Each storage node owns a subset of the volumes in a cell, each volume is owned by exactly one storage node at a time, and all volumes are owned at all times. The volume-to-storage-node assignment is maintained by a separate system that is out of the scope of this paper.

size—e.g., 40KB instead of 1GB—reconstructing the BLOB is much faster and lighter weight than rebuilding the block. Full block rebuilding is handled offline by rebuilder nodes.

Rebuilder Nodes At large scale, disk and node failures are inevitable. When this happens blocks stored on the failed components need to be rebuilt. Rebuilder nodes are storage-less, CPU-heavy nodes that handle failure detection and background reconstruction of data blocks. Each rebuilder node detects failure through probing and reports the failure to a coordinator node. It rebuilds blocks by fetching n companion or parity blocks from the failed block's strip and decoding them. Rebuilding is a heavyweight process that imposes significant I/O and network load on the storage nodes. Rebuilder nodes throttle themselves to avoid adversely impacting online user requests. Scheduling the rebuilds to minimize the likelihood of data loss is the responsibility of the coordinator nodes.

Coordinator Nodes A cell requires many maintenance task, such as scheduling block rebuilding and ensuring that the current data layout minimizes the chances of data unavailability. Coordinator nodes are storage-less, CPU-heavy nodes that handle these cell-wide tasks.

As noted earlier, blocks in a stripe are laid out on different failure domains to maximize reliability. However, after initial placement and after failure, reconstruction, and replacement there can be *violations* where a stripe's blocks are in the same failure domain. The coordinator runs a placement balancer process that validates the block layout in the cell, and rebalance blocks as appropriate. Rebalancing operations, like rebuilding operations, incur significant disk and network load on storage nodes and are also throttled so that user requests are adversely impacted.

5.4 Geo-replication

Individual f4 cells all reside in a single datacenter and thus are not tolerant to datacenter failures. To add datacenter fault tolerance we initially double-replicated f4 cells and placed the second replica in a different datacenter. If either datacenter fails, all the BLOBs are still available from the other datacenter. This provides all of our fault tolerance requirements and reduces the effective-replication-factor from 3.6 to 2.8.

Given the rarity of datacenter failure events we sought a solution that could further reduce the effective-replication-factor with the tradeoff of decreased throughput for BLOBs stored at the failed datacenter. We are currently deploying geo-replicated XOR coding that reduces the effective-replication-factor to 2.1.

Geo-replicated XOR coding provides datacenter fault tolerance by storing the XOR of blocks from two different volumes primarily stored in two different datacenters in a third datacenter as shown in Figure 9. Each data and parity block in a volume is XORed with the equivalent data or parity block in the other volume, called its *buddy block*, to create their *XOR block*. These XOR blocks are stored with normal triple-replicated index files for the volumes. Again, because the index files are tiny relative to the data, coding them is not worth the added complexity.

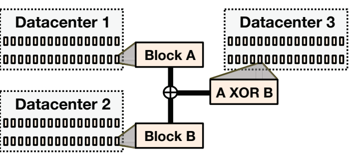


Figure 9: Geo-replicated XOR Coding.

The 2.1 replication factor comes from the 1.4X for the primary single cell replication for each of two volumes and another 1.4X for the geo-replicated XOR of the two volumes: $\frac{1.4*2+1.4}{2} = 2.1$.

Reads are handled by a *geo-backoff node* that receives requests for a BLOB that includes the data file, offset, and length (`R6` in Figure 8). This node then fetches the specified region from the local XOR block and the remote XOR-companion block and reconstructs the requested BLOB. These reads go through the normal single-cell read path through storage nodes Index and File APIs or backoff node File APIs if there are disk, host, or rack failures that affect the XOR or XOR-companion blocks.

We chose XOR coding for geo-replication because it significantly reduces our storage requirements while meeting our fault tolerance goal of being able to survive the failure of a datacenter.

5.5 f4 Fault Tolerance

Single f4 cells are tolerant to disk, host, and rack failures. Geo-replicating XOR volumes brings tolerance to datacenter failures. This subsection explains the failure domains in a single cell, how f4 lays out blocks to increase its resilience, gives an example of recovery if all four types of failure all affect the same BLOB, and summarizes how all components of a cell are fault tolerant.

Failure Domains and Block Placement Figure 10 illustrates how data blocks in a stripe are laid out in a f4 cell. A rack is the largest failure domain and is our primary concern. Given a stripe S of n data blocks and k parity blocks, we attempt to lay out the blocks so that each of these is on a different rack, and at least on a different node. This requires that a cell have at least $n + k$ racks, of roughly the same size. Our current implementation initially lays out blocks making a best-effort to put each on a different rack. The placement balancer process detects and corrects any rare violations that place a stripe's blocks on the same rack.

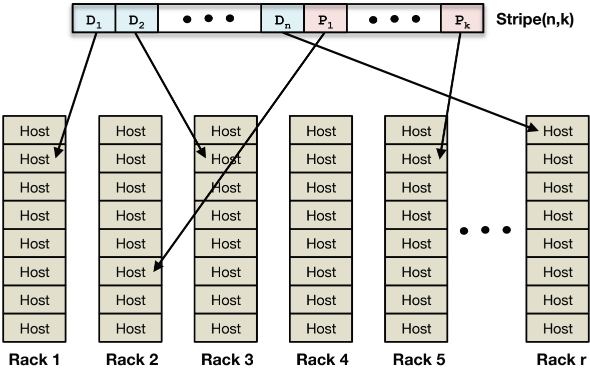


Figure 10: Distributing data & parity blocks in a f4 cell.

Laying blocks for a stripe out on different racks also provide resilience to host and disk failures. Blocks in a stripe on different racks will also be on different hosts and disks.

Quadruple Failure Example To illustrate f4's fault tolerance we examine a case where a failure at all four levels affects a single BLOB. The failures include:

1. Primary cell's datacenter fails.
2. Data block's XOR block's rack fails.
3. One of the parity block's XOR block's host fails.
4. Data block's XOR-companion block's disk fails.

The router tier will detect the primary's cell datacenter failure and send a BLOB read request to the XOR datacenter. The BLOB read request will be converted to a data file read request with an offset and length by the Index API on a geo-storage node using the triple-replicated index file in the XOR datacenter. Then a geo-backoff node will fetch the equivalent section of the XOR-data block locally and the buddy block from a third datacenter. The local XOR-data block read will initially fail because its enclosing rack is unavailable. Then the XOR-backoff node reads the XOR-data block through a (regular) backoff node that reconstructs the XOR-data block from n of its companion and parity blocks. Simultaneously, the remote buddy block read will fail because its enclosing disk failed. A (regular) backoff node in that datacenter will reconstruct the relevant section of buddy block from n of its companion and parity blocks. The XOR-backoff node will then receive the sections of the XOR-data block and the buddy block, XOR them, and return the BLOB.

Fault Tolerance for All Our primary fault tolerance design concern for f4 was providing four level of fault tolerance for data files, the dominant resource for warm BLOB storage, at a low effective-replication-factor. We also require that the other components of a cell be tolerance to the same faults, but use simpler and more

Node	Fault Tolerance Strategy
Name	Primary-backup; 2 backups; different racks.
Coordinator	"
Backoff	Soft state only.
Rebuilder	"
Storage:	
Index	3x local cell; 3x remote cell.
Data	Reed-Solomon local cell; XOR remote cell.

Table 1: Fault tolerance strategy for components of f4.

common techniques because they are not the dominant resource. Table 1 summarizes the techniques we use for fault tolerance for all components of a cell for failures within a cell. We do not provide datacenter fault tolerance for the other components of a cell because they are fate-sharing, i.e., datacenter failures take down entire cells.

5.6 Additional Design Points

This subsection briefly covers additional design points we excluded from the basic f4 design for clarity.

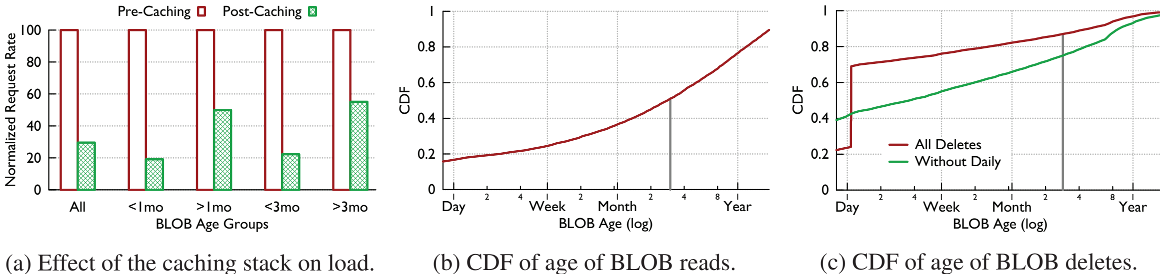
Mixing Age and Types Our BLOB storage system fills many volumes for each BLOB type concurrently. This mixes the age of BLOBs within a volume and smoothes their temperature. The most recent BLOBs in a volume may have a higher temperature than our target for f4. But, if the older BLOBs in the volume reduce its overall temperature below our target the volume may still be migrated to f4.

Different BLOB types are mixed together on hosts in a f4 cell to achieve a similar effect. High temperature types can be migrated to f4 sooner if they are mixed with low temperature types that will smooth out the overall load on each disk.

Index Size Consideration The memory needs of f4 (and Haystack) are primarily driven by the memory footprint of the index. The multiple caching layers in front of f4 obviate the need for a large buffer cache on the storage machine.[3]

Other than for profile photos, the memory sizes for the index fit into the memory in our custom hardware. For profile photos, we currently exclude them from f4 and keep them in Haystack. The index size for profile photos is still problematic for Haystack hosts, even though they store fewer BLOBs than f4 hosts. To keep the index size reasonable we under utilize the storage on the Haystack hosts. This enabled us to keep Haystack simple and does not significantly impact the efficiency of the overall system because there is only a single profile photo per user and they are quite small.

[3] A small buffer cache in Haystack is useful for newly written BLOBs, which are likely to be read and are not yet in the caching stack.



(a) Effect of the caching stack on load. (b) CDF of age of BLOB reads. (c) CDF of age of BLOB deletes.

Figure 11: Effects of our general architecture on the workload for f4.

Looking forward, we are evaluating lower-powered CPUs for the storage nodes because the CPU requirements are quite low. Unfortunately, lower powered CPUs usually come with smaller on-board memories. This, coupled with the fact that the drive densities as well as the number of drives per box are increasing, means that the index might not fit in memory for these lower-end configurations. We are exploring storing the index on flash instead of memory for these future configurations.

Software/Hardware Co-Design An important consideration in the design of f4 was keeping the hardware and software well matched. Hardware that provides capacity or IOPS that are not used by the software is wasteful; software designed with unrealistic expectations of the hardware will not work. The hardware and software components of f4 were co-designed to ensure they were well-matched by using software measurements to inform hardware choices and vice-versa.

For instance, we measured the candidate hard drives for f4 using a synthetic benchmark to determine the maximum IOPS we could consistently achieve while keeping per-request latency low. We then used these measurements to inform our choice of drives and our provisioning on the software side. The f4 software is designed so the weekly peak load on any drive is less than the maximum IOPS it can deliver.

6. Evaluation

This evaluation answers four key questions. Does our overall BLOB storage architecture enable warm storage? Can f4 handle the warm BLOB storage workload's throughput and latency requirements? Is f4 fault tolerant? And, does f4 save a significant amount of storage?

6.1 Methodology

Section 6.4 presents analytic results, all other results in this section are based on data captured from our production systems. The caching stack results in Section 6.2 are based on a day-long trace of 0.5% of BLOB requests routed through Facebook's caching stack; they do not include results served from browser or device caches. The read/delete results in Section 6.2 are based on a two-week sample from the router tier of 0.1% of reads and 10% of deletes. The results in Section 6.3 are obtained by dynamically tracking all operations to a uniform sample (0.01%) of all stored content. The storage savings in Section 6.5 are from measurements on a subset of f4.

We measure performance on our production system using a uniform sampling function so multiple generations of our storage machines are reflected in the captured data. Our older storage machines are commodity servers with a quad-core Intel Xeon CPU, 16/24/32 GB of memory, a hardware raid controller with 256-512 byte NVRAM and 12 x 1TB/2TB/3TB SATA drives. More recent machines are custom hosts with an Open Vault 2U chassis holding 30 x 3TB/4TB SATA drives [42]. Haystack uses Hardware RAID-6 with a NVRAM write-back cache while f4 uses these machines in a JBOD (Just a Bunch Of Disks) configuration.

6.2 General Architecture Enables Warm Storage

Our general architecture enables our warm storage system in four ways: (1) the caching stack reduces the load on f4; (2) the hot storage system bears the majority of reads and deletes, allowing our warm storage system to focus on efficient storage; (3) the router tier allows us to migrate volumes easily because it is an abstraction layer on top of the physical storage; and (4) the transformer tier allows an independent scaling of processing and storage.

The latter two points (3) and (4) are fundamental to our design. We validate points (1) and (2) experimentally.

Caching Stack Enables f4 Figure 11a shows the normalized request rate for BLOBs before and after the caching stack for different groups of BLOBs based on age. The Figure shows the caching stack reduces the request rate for all BLOBs to ~30% of what it would have otherwise been. Caching is the most effective for the most popular content, which we expect to be newer content. Thus, we expect the reduction in load from the cache to be less for older content. Our data shows this with the caching stack reducing the request rate to 3+ month old BLOBs to ~55% of its pre-caching volume. This reduction is still significant, however, without it the load for these BLOBs would increase $\frac{100-55}{55} = 82\%$.

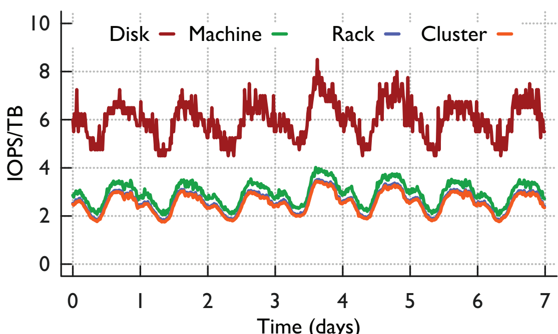


Figure 12: Maximum request rates over a week to f4's most loaded cluster.

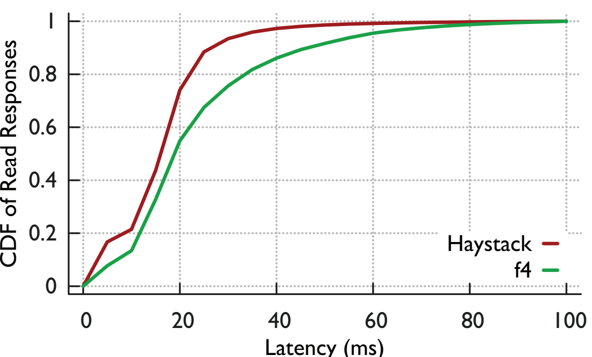


Figure 13: CDF of local read latency for Haystack/f4.

Haystack Enables f4 Figure 11b shows the CDF of the age of read BLOBs. Haystack handles all read requests for BLOBs less than 3 months old and some of the read request for BLOBs older than that.[4] This accounts for more than 50% of the read requests, significantly lowering the load on f4.

Figure 11c shows the CDF of the age of deleted BLOBs. All deletes are plotted, and all deletes excluding those for BLOBs that auto-expire after a day are plotted. Haystack again handles all deletes for content less than 3 months old. Haystack absorbs most BLOB deletes—over 70% of deletes excluding auto-expiry, and over 80% of deletes including auto-expiry—making them less of a concern for f4.

6.3 f4 Production Performance

This subsection characterizes f4's performance in production and demonstrated it can handle the warm storage workload and that it provides low latency for reads.

f4 Handles Peak Load The IOPS requirement for real-time requests is determined by the peak load rather than average requirement, so we need to look at peak request rates at a fine granularity. Figure 12 shows load in IOPS/TB for the f4 cluster with the highest load over the course of a week. The data is gathered from the 0.1% of reads trace and we compensate for the sampling rate by examining windows of 1000 seconds (instead of 1 second). Our trace identifies only the cluster for each request, so we randomly assign BLOBs to disks and use this assignment to determine the load on disks, machines, and racks. The maximum across all disk, machines, and racks is reported for each time interval.

The figure show the request rate has predictable peaks and troughs that result from different users across the globe accessing the site at different times and this can vary load by almost 2x during the course of a day.

[4] We currently use an approximately 3-month cutoff for all types in production for simplicity. BLOBs older than 3 months can be served by Haystack due to lag in migrations to f4.

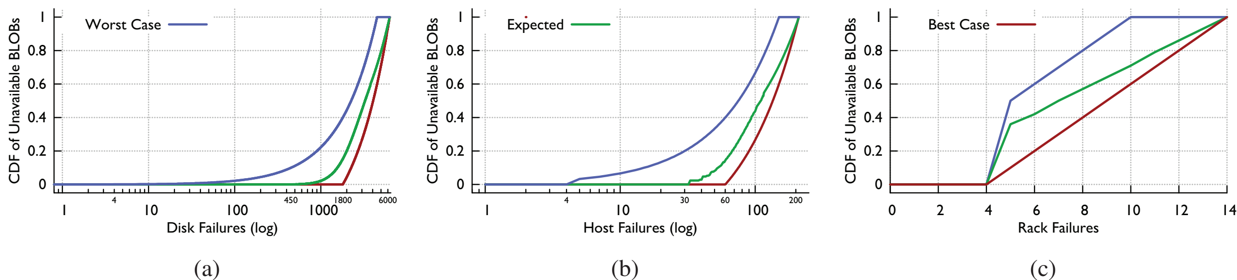
The maximum rack load is indistinguishable from the cluster load in the figure and peaks at 3.5 IOPS/TB during the trace week. The maximum machine load is slightly higher and peaks at 4.1 IOPS/TB. Maximum disk load is notably higher and peaks at 8.5 IOPS/TB. All of these are still less than half the 20 IOPS/TB maximum rate of f4. Even when examining the near-worse-case loads, f4 is able to cope with the lower throughput and decreased variance demands of warm BLOBs.

f4 Provides Low Latency Figure 13 shows the same region read latency for Haystack and f4. In our system, most (>99%) of the storage tier read accesses are within the same region. The latencies for f4 reads are higher than those for Haystack, e.g., the median read latency is 14 ms for Haystack and 17 ms for f4. But, the latency for f4 are still sufficiently low to provide a good user experience: the latency for reads in f4 is less than 30 ms for 80% of them and 80 ms for 99% of them.

6.4 f4 is Resilient to Failure

f4 is resilient to datacenter failures because we replicate data in multiple geographically distinct locations. Here we verify that f4 is resilient to disk, host, and rack failure.

Our implementation places blocks on different racks initially and continually monitors and rebalances blocks so they are on different racks due to failure. The result is that blocks are almost always in different failure domains, which we assume to be true for the rest of this analysis. Figure 14 shows the CDF of BLOBs that are unavailable if N disks, hosts, or racks fail in an f4 cell. Worst case, expected case, and best case CDFs are plotted. All results assume we lose some data when there are more than 4 failures in a stripe, though there is work that can recover some of this data [22] we do not implement it. Worst case results assume failures are assigned to one or a small number of blocks first and that parity blocks are the last to fail. Best case results assume failures are assigned to individual racks first and that parity blocks are the first to fail. Non-parity blocks can be used to individually extract the BLOBs they enclose. Expected results are calculated



(a) (b) (c)

Figure 14: Fraction of unavailable BLOBs for a f4 cell with N disk, host, and rack failures.

by the Monte Carlo method. There are 30 disks/host, 15 hosts/rack, and 14 racks.

Figure 14a shows the results for N disk failures. In the worst case there are some unavailable BLOBs after 4 disk failures, 50% unavailable BLOBs after 2250 disk failures, and 100% unavailable BLOBs after 4500 disk failures. In the best case there are no unavailable BLOBs until there are more than 1800 disk failures. In expectation, there will be some unavailable BLOBs after 450 disk failures, and 50% unavailable BLOBs after 3200 disk failures.

Figure 14b shows the results for N host failures. In the worst case there are unavailable BLOBs after 4 host failures, 50% unavailable BLOBs after 74 host failures, and 100% unavailable BLOBs after 150 host failures. In the best case, there are no unavailable BLOBs until there are more than 60 host failures. In expectation, there will be some unavailable BLOBs with 32 host failures and 50% unavailable BLOBs once there are 100 host failures.

Figure 14c shows the results for N rack failures. In the worst case there are unavailable BLOBs after 4 rack failures and 100% unavailable BLOBs after 10 rack failures. Even in the best case, there will be some unavailable BLOBs once there are 5 rack failures. In expectation, once there are 7 rack failures 50% of BLOBs will be unavailable. Taken together Figure 14 demonstrates that f4 is resilient to failure.

Failure Experience In general, we see an Annualized Failure Rate (AFR) of ~1% for our disks and they are replaced in less than 3 business days so we typically have at most a few disks out at a time per cluster. We recently received a batch of bad disks and have a higher failure rate for the cluster they are in, as discussed further in Section 7. Even so, we are always on the far left parts of the graphs in Figure 14 where there is no difference between worst/best/expected thus far. Host failures occur less often, though we do not have a rule-of-thumb failure rate for them. Host failures typically do not lose data, once the faulty component is replaced (e.g., DRAM) the host returns with the data still on its disks. Our worst failure thus far has been a self-inflicted drill that rebuilt 2 hosts worth of data (240 TB) in the background over 3 days. The only adverse affect of the drill was an increase in p99 latency to 500ms.

6.5 f4 Saves Storage

f4 saves storage space by reducing the effective-replication-factor of BLOBs, but it does not reclaim the space of deleted BLOBs. Thus, the true benefit in reduced storage for f4 must account for the space. We measured the space used for deleted data in f4, which was 6.8%.

Let $repl_{\text{hay}} = 3.6$ be the effective replication factor for Haystack, $repl_{\text{f4}} = 2.8$ or 2.1 be the effective replication factor of f4, $del_{\text{f4}} = .068$ the fraction of BLOBs in f4 that are deleted, and $logical_{\text{f4}} > 65PB$ be the logical size of BLOBs stored in f4. Then the reduction in storage space from f4 is:

$$\begin{aligned}\text{Reduction} &= (repl_{\text{hay}} - repl_{\text{f4}} * \frac{1}{1 - del_{\text{f4}}}) * logical_{\text{warm}} \\ &= (3.6 - repl_{\text{f4}} * 1.07) * 65\text{PB} \\ &= 30\text{PB } at\ 2.8, 68\text{PB } at\ 2.1, 53\text{PB currently}\end{aligned}$$

With a current corpus over 65 PB, f4 saved over 39 PB of storage at the 2.8 effective-replication-factor and will save over 87 PB of storage at 2.1. f4 currently saves over 53PB with the partial deployment of 2.1.

7. Experience

In the course of designing, building, deploying, and refining f4 we learned many lessons. Among these the importance of simplicity for operational stability, the importance of measuring underlying software for your use case's efficiency, and the need for heterogeneity in hardware to reduce the likelihood of correlated failures stand out.

The importance of simplicity in the design of a system for keeping its deployment stable crops up in many systems within Facebook [41] and was reinforced by our experience with f4. An early version of f4 used journal files to track deletes in the same way that Haystack does. This single read-write file was at odds with the rest of the f4 design, which is read-only. The at-most-one-writer requirement of the distributed file system at the heart of

our implementation (HDFS), the inevitability of failure in large distributed systems, and the rarity of writes to the journal file did not play well together. This was the foremost source of production issues for f4. Our later design that removed this read-write journal file pushed delete tracking to another system that was designed to be read-write. This change simplified f4 by making it fully read-only and fixed the production issues.

Measuring and understanding the underlying software that f4 was built on top of helped improve the efficiency of f4. f4's implementation is built on top of the Hadoop File System (HDFS). Reads in HDFS are typically handled by any server in the cluster and then proxied by that server to the server that has the requested data. Through measurement we found that this proxied read has lower throughput and higher latency than expected due to the way HDFS schedules IO threads. In particular, HDFS used a thread for each parallel network IO request and Java's multithreading did not scale well to a large number of parallel requests, which resulted in an increasing backlog of network IO requests. We worked around this with a two-part read, described in Section 5.3, that avoids proxying the read through HDFS. This workaround resulted in the expected throughput and latency for f4.

We recently learned about the importance of heterogeneity in the underlying hardware for f4 when a crop of disks started failing at a higher rate than normal. In addition, one of our regions experienced higher than average temperatures that exacerbated the failure rate of the bad disks. This combination of bad disks and high temperatures resulted in an increase from the normal ~1% AFR to an AFR over 60% for a period of weeks. Fortunately, the high-failure-rate disks were constrained to a single cell and there was no data loss because the buddy and XOR blocks were in other cells with lower temperatures that were unaffected. In the future we plan on using hardware heterogeneity to decrease the likelihood of such correlated failures.

8. Related Work

We divide related work into distributed file system, distributed disk arrays, erasure codes, erasure coded storage, hierarchical storage, other related techniques, and BLOB storage systems. f4 is primarily distinguished by its specificity and thus simplicity, and by virtue of it running in production at massive scale across many disk, hosts, racks, and datacenters.

Distributed File Systems There are many classic distributed file systems including Cedar [26], Andrew [32], Sprite [39], Coda [48], Harp [38], xfs [3], and Petal [36] among many others. Notable recent examples include the Google File System [18], BigTable [12], and Ceph [53]. All of these file systems are much more general, and thus necessarily more complex, than f4 whose design was informed by its simpler workload.

Distributed Disk Arrays There is also a large body of work on striping data across multiple disks for improved throughput and fault tolerance that was first advocated in a case for RAID [43]. Later work included Zebra [30] that forms of a client's write into a log and stripes them together, similar to how we stripe many BLOBs together in a block. Other work includes disk shadowing [7], maximizing performance in a striped disk array [13], parity declustering [31], parity logging [51], AFRAID [49], TickerTAIP [11], NASD [19], and D-GRAID [50]. Chen et al.'s survey on provides a thorough overview of RAID in practice [14]. f4 continues the tradition of distributing data for reliability, but does so across racks and datacenter as well as disks and hosts.

Erasure Codes Erasure codes enjoy a long history starting with the Hamming's original error-correcting code [27]. Our work uses Reed-Solomon codes [46] and XOR codes. EVENODD [8] simplifies error correction using XOR codes. WEAVER codes [24] are a more recent XOR-based erasure code. HoVer codes [25] add parity in two dimensions, similar to our local vs. geo-replicated distinction, though at a much lower level and with more similar techniques. STAIR codes [37] provide fault tolerance to disk sector failures, a level below our currently smallest failure domain. XORing elephants [4] presents a new family of erasure codes that are more efficiently repairable. A hitchhiker's guide to fast and efficient data reconstruction [45] presents new codes that reduce network and disk usage. f4 uses erasure codes as tools and does not innovate in this area.

Erasure Coded Storage Plank gave a tutorial on Reed-Solomon codes for error correction in RAID-like systems [44]. f4 implements something similar, but uses checksums colocated with blocks for error detection and uses Reed-Solomon for erasure correction that can tolerate more failures at same parity level. More recent erasure coded storage includes Oceanstore [35], a peer-to-peer erasure coded system. Weatherspoon et al. [52] provide a detailed comparison of replication vs. erasure-coding for peer-to-peer networks. Other systems include Glacier [23] and Ursa Minor [1]. Windows Azure storage [33] uses new Local Reconstruction Codes for efficient recovery with local and global parity information, but is not a Maximum Distance Separable (MDS) code. Our local Reed-Solomon coding is MDS, though the combination with XOR is not.

Hierarchical Storage The literature is also rich with work on hierarchical storage that uses different storage subsystems for different working sets. A canonical example is HP AutoRAID [54] that has two levels of storage with replication at the top-level and RAID 5 for the bot-

tom level. HP AutoRAID transparently migrates data between the two levels based on inactivity. The replication our BLOB storage system is similar, though at a far larger scale, for a simpler workload, and with very different migration choices and costs.

Other Similar Techniques Our approach of appending new BLOBs to a physical volume resembles log-structured file systems [47], and greatly improves our write latency. Harter et al. [28] analyzed the I/O behavior of *iBench*, a collection of productivity and multimedia applications and observed that many modern applications manage a single file as a mini-filesystem. This is also how we treat our files (including data files, index files and journal files). Copyset replication [15] explores how to group replicas to decrease the likelihood of data loss, but does not use erasure codes.

BLOB Storage Our work on warm storage builds on some key ideas from Haystack [5], Facebook's hot BLOB storage system. Huang et al. [34] performed an extensive study of Facebook photo and found that advanced caching algorithms would increase cache hit ratios and further drive down backend load. If implemented, this could enable faster migration from hot storage to f4. Harter et al. [29] performed a multilayer study of the Facebook Messages stack, which is also built on top of HDFS. Blobstore [6] provides a good overview of Twitter's in-house photo storage system, but does not describe performance or efficiency aspects in much detail. Microsoft's Windows Azure Storage [10] is designed to be a generic cloud service while ours is a more specialized application, with more unique challenges as well as optimization opportunities. Their coding techniques are discussed above. Google provides a durable but reduced availability storage service (DRA) on its cloud platform [21], but implementation details are not public and there is no support for migrating groups of objects (buckets) from normal to DRA storage.

9. Conclusion

Facebook's BLOB corpus is massive, growing, and increasingly warm. This paper made a case for a specialized warm BLOB storage system, described an overall BLOB storage system that enables warm storage, and gave the design of f4. f4 reduces the effective-replication-factor of warm BLOBs from 3.6 to 2.1; is fault tolerant to disk, host, rack, and datacenter failures; provides low latency for client requests; and is able to cope with the lower throughput demands of warm BLOBs.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: Versatile cluster-based storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[2] Akamai. `http://www.akamai.com`.

[3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[4] M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. *Proceedings of the VLDB Endowment (PVLDB)*, 6(5), 2013.

[5] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[6] A. Bigian. Blobstore: Twitter's in-house photo storage system. `http://tinyurl.com/cda5ahq`, 2012.

[7] D. Bitton and J. Gray. Disk shadowing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1988.

[8] M. Blaum, J. Brady, J. Bruck, and J. Menon. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Transactions on Computers*, 44 (2), 1995.

[9] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2013.

[10] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[11] P. Cao, S. B. Lin, S. Venkataraman, and J. Wilkes. The tickertaip parallel raid architecture. *ACM Transactions on Computer Systems (TOCS)*, 12(3), 1994.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions of Computer Systems (TOCS)*, 26(2), 2008.

[13] P. M. Chen and D. A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 1990.

[14] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), 1994.

[15] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2013.

[16] Facebook Groups. Facebook groups. https://www.facebook.com/help/groups.

[17] Facebook Mobile Photo Sync. Facebook mobile photo sync. https://www.facebook.com/help/photosync.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP)*, 2003.

[19] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *ACM SIGPLAN Notices*, 1998.

[20] A. Glacier. https://aws.amazon.com/glacier/.

[21] Google. Durable reduced availability storage. https://developers.google.com/storage/docs/durable-reduced-availability, 2014.

[22] V. Guruswami and M. Sudan. Improved decoding of reed-solomon and algebraic-geometry codes. *IEEE Transactions on Information Theory*, 1999.

[23] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the Conference on Symposium on Networked Systems Design & Implementation (NSDI)*, 2005.

[24] J. L. Hafner. Weaver codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[25] J. L. Hafner. Hover erasure codes for disk arrays. In *International Conference on Dependable Systems and Networks (DSN)*, 2006.

[26] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1987.

[27] R. W. Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2), 1950.

[28] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proc. Symposium on Operating Systems Principles (SOSP)*, 2011.

[29] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[30] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3), 1995.

[31] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[32] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1), 1988.

[33] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in windows azure storage. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2012.

[34] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[35] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35 (11), 2000.

[36] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Architectural support for programming languages and operating systems (ASPLOS)*, 1996.

[37] M. Li and P. P. C. Lee. Stair codes: A general family of erasure codes for tolerating device and sector failures in practical storage systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[38] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[39] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Transaction on Computer Systems (TOCS)*, 6(1), 1988.

[40] J. Niccolai. Facebook puts 10,000 blu-ray discs in low-power storage system. http://tinyurl.com/qx759f4, 2014.

[41] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[42] Open Compute. Open compute. http://www.opencompute.org/.

[43] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, 1988.

[44] J. S. Plank et al. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Software: Practice and Experience*, 27(9), 1997.

[45] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhikers guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the ACM conference on SIGCOMM*, 2014.

[46] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8, 1960.

[47] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions of Computer Systems (TOCS)*, 10(1), 1992.

[48] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), 1990.

[49] S. Savage and J. Wilkes. Afraid: A frequently redundant array of independent disks. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 1996.

[50] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with d-graid. *ACM Transactions on Storage (TOS)*, 1(2), 2005.

[51] D. Stodolsky, G. Gibson, and M. Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[52] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International workshop on Peer-To-Peer Systems (IPTPS*. 2002.

[53] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *In Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[54] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The hp autoraid hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1), 1996.

SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi*,
Jeffrey F. Lukman† and Haryadi S. Gunawi

University of Chicago *NEC Labs America †Surya University

Abstract

The last five years have seen a rise of implementation-level distributed system model checkers (dmck) for verifying the reliability of real distributed systems. Existing dmcks however rarely exercise multiple failures due to the state-space explosion problem, and thus do not address present reliability challenges of cloud systems in dealing with complex failures. To scale dmck, we introduce semantic-aware model checking (SAMC), a white-box principle that takes simple semantic information of the target system and incorporates that knowledge into state-space reduction policies. We present four novel reduction policies: local-message independence (LMI), crash-message independence (CMI), crash recovery symmetry (CRS), and reboot synchronization symmetry (RSS), which collectively alleviate redundant reorderings of messages, crashes, and reboots. SAMC is systematic; it does not use randomness or bug-specific knowledge. SAMC is simple; users write protocol-specific rules in few lines of code. SAMC is powerful; it can find deep bugs one to three orders of magnitude faster compared to state-of-the-art techniques.

1 Introduction

As more data and computation move from local to cloud settings, cloud systems[1] such as scale-out storage systems [7, 13, 18, 41], computing frameworks [12, 40], synchronization services [5, 28], and cluster management services [27, 47] have become a dominant backbone for many modern applications. Client-side software is getting thinner and more heavily relies on the capability, reliability, and availability of cloud systems. Unfortunately, such large-scale distributed systems remain difficult to get right. Guaranteeing reliability has proven to be challenging in these systems [23, 25, 51].

Software (implementation-level) model checking is one powerful method of verifying systems reliability [21, 52, 53]. The last five years have seen a rise of software model checkers targeted for distributed systems [22, 25, 43, 50, 51]; for brevity, we categorize such systems as *dmck* (distributed system model checker). Dmck works by exercising all possible sequences of events (*e.g.*, different reorderings of messages), and hereby pushing the target system into corner-case situations and unearthing hard-to-find bugs. To address the state-space explosion problem, existing dmcks adopt advanced state reduction techniques such as dynamic partial order reduction (DPOR), making them mature and highly practical for checking large-scale systems [25, 51].

Despite these early successes, existing dmcks unfortunately fall short in addressing present reliability challenges of cloud systems. In particular, large-scale cloud systems are expected to be highly reliable in dealing with complex failures, not just one instance, but *multiple* of them. However, to the best of our knowledge, *no* existing dmcks can exercise multiple failures without exploding the state space. We elaborate this issue later; for now, we discuss complex failures in cloud environments.

Cloud systems run on large clusters of unreliable commodity machines, an environment that produces a growing number and frequency of failures, including "surprising" failures [2, 26]. Therefore, it is common to see complex failure-induced bugs such as the one below.

> **ZooKeeper Bug #335:** **(1)** Nodes A, B, C start with latest txid #10 and elect B as leader, **(2)** *B crashes*, **(3)** Leader election re-run; C becomes leader, **(4)** Client writes data; A and C commit new txid-value pair {#11:X}, **(5)** *A crashes before* committing tx #11, **(6)** C loses quorum, **(7)** *C crashes*, **(8)** *A reboots* and *B reboots*, **(9)** A becomes leader, **(10)** Client updates data; A and B commit a new txid-value pair {#11:Y}, **(11)** *C reboots after* A's new tx commit, **(12)** C synchronizes with A; C notifies A of {#11:X}, **(13)** A replies to C the "diff" starting with tx 12 (excluding tx {#11:Y}!), **(14)** Violation: permanent data inconsistency as A and B have {#11:Y} and C has {#11:X}.

The bug above is what we categorize as *deep bug*. To unearth deep bugs, dmck must permute a large number

[1] These systems are often referred with different names (*e.g.*, cloud software infrastructure, datacenter operating systems). For simplicity, we use the term "cloud systems".

of events, not only network events (messages), but also *crashes* and *reboots*. Although arguably deep bugs occur with lower probabilities than "regular" bugs, deep bugs do occur in large-scale deployments and have harmful consequences (§2.3). We observe that cloud developers are prompt in fixing deep bugs (in few weeks) as they seem to believe in Murphy's law; at scale, anything that can go wrong will go wrong.

As alluded above, the core problem is that state-of-the-art dmcks [22, 25, 34, 43, 50, 51] do not incorporate failure events to their state exploration strategies. They mainly address scalability issues related to message re-orderings. Although some dmcks are capable of injecting failures, usually they only exercise at most one failure. The reason is simple: exercising crash/reboot events will exacerbate the state-space explosion problem. In this regard, existing dmcks do not scale and take very long time to unearth deep bugs. This situation led us to ask: *how should we advance dmck to discover deep bugs quickly and systematically, and thereby address present reliability challenges of cloud systems in dealing with complex failures?*

In this paper, we present semantic-aware model checking (SAMC; pronounced "Sam-C"), a white-box principle that takes simple semantic information of the target system and incorporates that knowledge in state-space reduction policies. In our observation, existing dmcks treat every target system as a complete black box, and therefore many times perform message re-orderings and crash/reboot injections that lead to the same conditions that have been explored in the past. These *redundant executions* must be removed significantly to tame the state-space explosion problem. We find that simple semantic knowledge can scale dmck greatly.

The main challenge of SAMC is in defining *what* semantic knowledge can be valuable for reduction policies and *how* to extract that information from the target system. We find that useful semantic knowledge can come from *event processing semantic* (*i.e.*, how messages, crashes, and reboots are processed by the target system). To help testers extract such information from the target system, we provide *generic event processing patterns*, patterns of how messages, crashes, and reboots are processed by distributed systems in general.

With this method, we introduce four novel semantic-aware reduction policies. First, *local-message independence* (LMI) reduces re-orderings of concurrent intra-node messages. Second, *crash-message independence* (CMI) reduces re-orderings of crashes among outstanding messages. Third, *crash recovery symmetry* (CRS) skips crashes that lead to symmetrical recovery behaviors. Finally, *reboot synchronization symmetry* (RSS) skips reboots that lead to symmetrical synchronization actions. Our reduction policies are *generic*; they are applicable to many distributed systems. SAMC users (*i.e.*, testers) only need to feed the policies with short *protocol-specific rules* that describe event independence and symmetry specific to their target systems.

SAMC is purely systematic; it does not incorporate randomness or bug-specific knowledge. Our policies run on top of sound model checking foundations such as state or architectural symmetry [9, 45] and independence-based dynamic partial order reduction (DPOR) [17, 20]. Although these foundations have been around for a decade or more, its application to dmck is still limited; these foundations require testers to define *what* events are actually independent or symmetrical. With SAMC, we can define fine-grained independence and symmetry.

We have built a prototype of SAMC (SAMPRO) from scratch for a total of 10,886 lines of code. We have integrated SAMPRO to three widely popular cloud systems, ZooKeeper [28], Hadoop/Yarn [47], and Cassandra [35] (old and latest stable versions; 10 versions in total). We have run SAMPRO on 7 different protocols (leader election, atomic broadcast, cluster management, speculative execution, read/write, hinted handoff, and gossiper). The protocol-specific rules are written in only 35 LOC/protocol on average. This shows the simplicity of applying SAMC reduction policies across different systems and protocols; all the rigorous state exploration and reduction are automatically done by SAMPRO.

To show the power of SAMC, we perform an extensive evaluation of SAMC's speed in finding deep bugs. We take 12 old real-world deep bugs that require multiple crashes and reboots (some involve as high as 3 crashes and 3 reboots) and show that SAMC can find the bugs one to three orders of magnitude faster compared to state-of-the-art techniques such as black-box DPOR, random+DPOR, and pure random. We show that this speed saves tens of hours of testing time. More importantly, some deep bugs cannot be reached by non-SAMC approaches, even after 2 days; here, SAMC's speed-up factor is potentially much higher. We also found 2 new bugs in the latest version of ZooKeeper and Hadoop.

To the best of our knowledge, our work is the first solution that systematically scales dmck with the inclusion of failures. We believe none of our policies have been introduced before. Our prototype is also the first available dmck for our target systems. Overall, we show that SAMC can address deep reliability challenges of cloud systems by helping them discover deep bugs faster.

The rest of the paper is organized as follows. First, we present a background and an extended motivation (§2). Next, we present SAMC and our four reduction policies (§3). Then, we describe SAMPRO and its integration to cloud systems (§4). Finally, we close with evaluations (§5), related work (§7), and conclusion (§8).

2 Background

This section gives a quick background on dmck and related terms, followed by a detailed overview of the state of the art. Then, we present cases of deep bugs and motivate the need for dmck advancements.

2.1 DMCK Framework and Terms

As mentioned before, we define *dmck* as software model checker that checks distributed systems directly at the implementation level. Figure 1 illustrates a dmck integration to a target distributed system, a simple representation of existing dmck frameworks [25, 34, 43, 51]. The dmck inserts an interposition layer in each node of the target system with the purpose of controlling all important events (*e.g.*, network messages, timeouts) and preventing the target system to process the events until the dmck enables them. A main dmck mechanism is the permutation of events; the goal is to push the target system into all possible ordering scenarios. For example, the dmck can enforce `abcd` ordering in one execution, `bcad` in another, and so on.

We now provide an overview of basic dmck terms we use in this paper and Figure 1. Each node of the target system has a *local state* (`ls`), containing many variables. An *abstract local state* (`als`) is a subset of the local state; dmck decides which `als` is important to check. The collection of all (and abstract) local states is the *global state* (`gs`) and the *abstract global state* (`ags`) respectively. The *network state* describes all the *outstanding messages* currently intercepted by dmck (*e.g.*, `abd`). To model check a specific protocol, dmck starts a *workload driver* (which restarts the whole system, runs specific workloads, etc.). Then, dmck generates many (typically hundreds/thousands) executions; an *execution* (or a *path*) is a specific ordering of events that dmck enables (*e.g.*, `abcd`, `dbca`) from an initial state to a termination point. A *sub-path* is a subset of a path/execution. An *event* is an action by the target system that is intercepted by dmck (*e.g.*, a network message) or an action that dmck can inject (*e.g.*, a crash/reboot). Dmck enables one event at a time (*e.g.*, `enable(c)`). To permute events, dmck runs *exploration methods* such as brute-force (*e.g.*, depth first search) or random. As events are permuted, the target system enters hard-to-reach states. Dmck continuously runs state *checks* (*e.g.*, safety checks) to verify the system's correctness. To reduce the state-space explosion problem, dmck can employ *reduction policies* (*e.g.*, DPOR or symmetry). A policy is *systematic* if it does not use randomness or bug-specific knowledge. In this work, we focus on advancing systematic reduction policies.

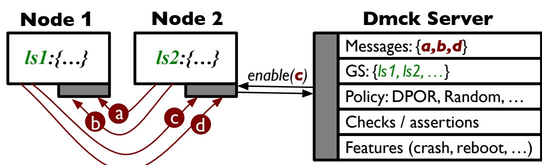


Figure 1: **DMCK.** *The figure illustrates a typical framework of a distributed system model checker (dmck).*

2.2 State-of-the-Art DMCKs

MODIST [51] is arguably one of the most powerful dmcks that comes with systematic reduction policies. MODIST has been integrated to real systems due to its exploration scalability. At the heart of MODIST is *dynamic partial order reduction (DPOR)* [17] which exploits the *independence* of events to reduce the state explosion. Independent events mean that it does not matter in what order the system execute the events, as their different orderings are considered equivalent.

To illustrate how MODIST adopts DPOR, let's use the example in Figure 1, which shows four concurrent outstanding messages `abcd` (`a` and `b` for `N1`, `c` and `d` for `N2`). A brute-force approach will try all possible combinations (`abcd`, `abdc`, `acbd`, `acdb`, `cabd`, and so on), for a total of 4! executions. Fortunately, the notion of event independence can be mapped to distributed system properties. For example, MODIST specifies this reduction policy: a message to be processed by a given node is independent of other concurrent messages destined to other nodes (based on vector clocks). Applying this policy to the example in Figure 1 implies that `a` and `b` are dependent[1] but they are independent of `c` and `d` (and vice versa). Since only dependent events need to be reordered, this reduction policy leads to only 4 executions (`ab-cd`, `ab-dc`, `ba-cd`, `ba-dc`), giving a 6x speed-up (4!/4).

Although MODIST's speed-up is significant, we find that one scalability limitation of its DPOR application is within its *black-box* approach; it only exploits general properties of distributed systems to define message independence. It does not exploit any semantic information from the target system to define more independent events. We will discuss this issue later (§3.1).

Dynamic interface reduction (DIR) [25] is the next advancement to MODIST. This work suggests that a complete dmck must re-order not only messages (global events) but also thread interleavings (local events). The reduction intuition behind DIR is that different thread interleavings often lead to the same global events (*e.g.*, a node sends the same messages regardless of how threads are interleaved in that node). DIR records local explo-

[1] In model checking, "dependent" events mean that they must be re-ordered. "Dependent" does not mean "causally dependent".

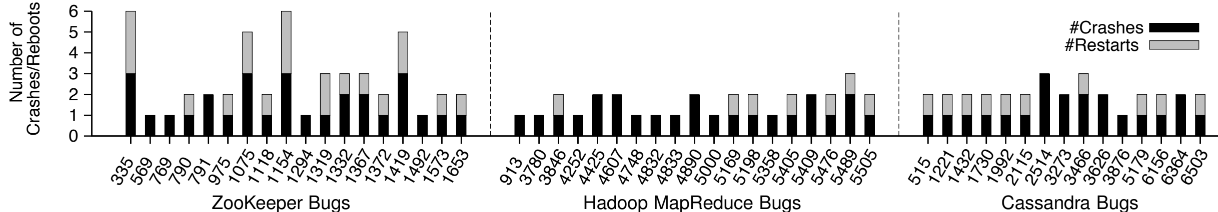


Figure 2: **Deep Bugs.** *The figure lists deep bugs from our bug study and depicts how many crashes and reboots must happen to reproduce the bugs. Failure events must happen in a specific order in a long sequence of events. These bugs came from many protocols including ZooKeeper leader election and atomic broadcast, Hadoop MapReduce speculative execution, job/task trackers, and resource/application managers, and Cassandra gossiper, anti-entropy, mutation, and hinted handoff. These bugs led to failed jobs, node unavailability, data loss, inconsistency, and corruption. They were labeled as "major", "critical", or "blocker". 12 of these bugs happened within the last one year. The median response time (i.e., time to fix) is two weeks. There are few bugs that involve 4+ reboots and 4+ crashes that we do not show here.*

ration and replays future incoming messages without the need for global exploration. In our work, SAMC focuses only on global exploration (message and failure re-orderings). We believe DIR is orthogonal to SAMC, similar to the way DIR is orthogonal to MODIST.

MODIST and DIR are examples of dmcks that employ advanced systematic reduction policies. LMC [22] is similar to DIR; it also decouples local and global exploration. dBug [43] applies DPOR similarly to MODIST. There are other dmcks such as MACEMC [34] and CrystalBall [50] that use basic exploration methods such as depth first (DFS), weight-based, and random searches.

Other than the aforementioned methods, *symmetry* is another foundational reduction policy [16, 45]. Symmetry-based methods exploit the architectural symmetry present in the target system. For example, in a ring of nodes, one can rotate the ring without affecting the behavior of the system. Symmetry is powerful, but we find no existing dmcks that adopt symmetry.

Besides dmcks, there exists sophisticated testing frameworks for distributed systems (*e.g.*, FATE [23], PREFAIL [31], SETSUDO [30], OpenStack fault-injector [32]). This set of work emphasizes the importance of multiple failures, but their major limitation is that they are not a dmck. That is, they cannot systematically control and permute non-deterministic choices such as message and failure reorderings.

2.3 Deep Bugs

To understand the unique reliability challenges faced by cloud systems, we performed a study of reliability bugs of three popular cloud systems: ZooKeeper [28], Hadoop MapReduce [47], and Cassandra [35]. We scanned through thousands of issues from their bug repositories. We then tagged complex reliability bugs that can only be caught by a dmck (*i.e.*, bugs that can occur only on specific orderings of events). We found 94 dmck-catchable bugs.[1] Our major finding is that 50% of them are deep bugs (require complex re-ordering of not only messages but also crashes and reboots).

Figure 2 lists the deep bugs found from our bug study. Many of them were induced by multiple crashes and reboots. Worse, to reproduce the bugs, crash and reboot events must happen in a specific order within a long sequence of events (*e.g.*, the example bug in §1). Deep bugs lead to harmful consequences (*e.g.*, failed jobs, node unavailability, data loss, inconsistency, corruption), but they are hard to find. We observe that since there is no dmck that helps in this regard, deep bugs are typically found in deployment (via logs) or manually, then they get fixed in few weeks, but afterwards as code changes continuously, new deep bugs tend to surface again.

2.4 Does State of the-Art Help?

We now combine our observations in the two previous sections and describe why state-of-the-art dmcks do not address present reliability challenges of cloud systems.

First, *existing systematic reduction policies often cannot find bugs quickly*. Experiences from previous dmck developments suggest that significant savings from sound reduction policies do not always imply high bug-finding effectiveness [25, 51]. To cover deep states and find bugs, many dmcks revert to non-systematic methods such as randomness or manual checkpoints. For example, MODIST combines DPOR with random walk to "jump" faster to a different area of the state space (§4.5 of [51]). DIR developers find new bugs by manually setting "interesting" checkpoints so that future state explorations happen from the checkpoints (§5.3 of [25]). In our work, although we use different target systems, we are able to reproduce the same experiences above (§5.1).

[1] Since this is a manual effort, we might miss some bugs. We also do not report "simple" bugs (*e.g.*, error-code handling) that can be caught by unit tests.

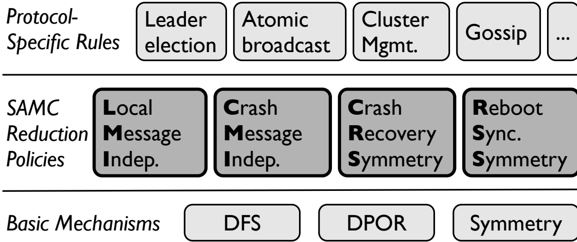


Figure 3: **SAMC Architecture.**

Second, *existing dmcks do not scale with the inclusion of failure events*. Given the first problem above, exercising multiple failures will just exacerbate the state-space explosion problem. Some frameworks that can explore multiple failures such as MACEMC [34] only do so in a random way; however, in our experience (§5.1), randomness many times cannot find deep bugs quickly. MODIST also enabled only one failure. In reality, multiple failures is a big reliability threat, and thus must be exercised.

We conclude that finding systematic (no random/checkpoint) policies that can find deep bugs is still an open dmck research problem. We believe without semantic knowledge of the target system, dmck hits a scalability wall (as also hinted by DIR authors; §8 of [25]). In addition, as crashes and reboots need to be exercised, we believe recovery semantics must be incorporated into reduction policies. All of these observations led us to SAMC, which we describe next.

3 SAMC

Semantic-aware model checking (SAMC) is a white-box model checking approach that takes semantic knowledge of how events (*e.g.*, messages, crashes, and reboots) are processed by the target system and incorporates that information in reduction policies. To show the intuition behind SAMC, we first give an example of a simple leader election protocol. Then, we present SAMC architecture and our four reduction policies.

3.1 An Example

In a simple leader election protocol, every node broadcasts its vote to reach a quorum and elect a leader. Each node begins by voting for itself (*e.g.*, `N2` broadcasts `vote=2`). Each node receives vote broadcasts from other peers and processes every vote with this simplified code segment below. As depicted in the code segment below, if an incoming vote is less than the node's current vote, it is simply discarded. If it is larger, the node changes its vote and broadcasts the new vote.

```
if (msg.vote < myVote) {discard;}
else {myVote = msg.vote; broadcast(myVote);}
```

Let's assume `N4` with `vote=4` is receiving three concurrent messages with votes `1`, `2`, and `3` from its peers. Here, a dmck with a black-box DPOR approach must perform 6 (3!) orderings (`123`, `132`, and so on). This is because a black-box DPOR does *not* know the *message processing semantic* (*i.e.*, how messages will be processed by the receiving node). Thus, a black-box DPOR must treat all of them as dependent (§2.2); they must be re-ordered for soundness. However, by knowing the processing logic above, a dmck can soundly conclude that all orderings will lead to the same state; all messages will be discarded by `N4` and its local state will not change. Thus, a semantic-aware dmck can reduce the 6 redundant executions to just 1 execution.

The example above shows a scalability limitation of a black-box dmck. Fortunately, simple semantic knowledge has a great potential in removing redundant executions. Furthermore, semantic knowledge can be incorporated on top of sound model checking foundations such as DPOR and symmetry, as we describe next.

3.2 Architecture

Figure 3 depicts the three levels of SAMC: sound exploration mechanisms, reduction policies, and protocol-specific rules. SAMC is built on top of sound model checking foundations such as DPOR [17, 20] and symmetry [9, 45]. We name these foundations as mechanisms because a dmck must specify accordingly what events are dependent/independent and symmetrical, which in SAMC will be done by the reduction policies and protocol-specific rules.

Our main contribution lies within our four novel *semantic-aware reduction policies*: local-message independence (LMI), crash-message independence (CMI), crash recovery symmetry (CRS), and reboot synchronization symmetry (RSS). To the best of our knowledge, none of these approaches have been introduced in the literature. At the heart of these policies are *generic event processing patterns* (*i.e.*, patterns of how messages, crashes, and reboots are processed by distributed systems). Our policies and patterns are simple and powerful; they can be applied to many different distributed systems. Testers can extract the patterns from their target protocols (*e.g.*, leader election, atomic broadcast) and write protocol-specific rules in few lines of code.

In the next section, we first present our four reduction policies along with the processing patterns. Later, we will discuss ways to extract the patterns from target systems (§3.4) and then show the protocol-specific rules for our target systems (§4.2).

3.3 Semantic-Aware Reduction Policies

We now present four semantic-aware reduction policies that enable us to define fine-grained event dependency/independency and symmetry beyond what black-box approaches can do.

3.3.1 Local-Message Independence (LMI)

We define *local messages* as messages that are concurrently in flight to a given node (*i.e.*, intra-node messages). As shown in Figure 4a, a black-box DPOR treats the message processing semantic inside the node as a black box, and thus must declare the incoming messages as dependent, leading to 4! permutation of `abcd`. On the other hand, with white-box knowledge, local-message independence (LMI) can define *independency relationship among local messages*. For example, illustratively in Figure 4b, given the node's local state (`ls`) and the processing semantic (embedded in the `if` statement), LMI is able to define that `a` and `b` are dependent, `c` and `d` are dependent, but the two groups are independent, which then leads to only 4 re-orderings. This reduction illustration is similar to the one in Section 2.2, but this time LMI enables DPOR application on local messages.

LMI can be easily added to a dmck. A dmck server typically has a complete view of the local states (§2.1). What is needed is the *message processing semantic*: how will a node (`N`) process an incoming message (`m`) given the node's current local state (`ls`)? The answer lies in these four simple *message processing patterns* (discard, increment, constant, and modify):

```
Discard:              Increment:
if (pd(m,ls))         if (pi(m,ls))
 (noop);                ls++;

Constant:             Modify:
if (pc(m,ls))         if (pm(m,ls))
  ls = Const;           ls = modify(m,ls);
```

In practice, `ls` and `m` contain many fields. For simplicity, we treat them as integers. The functions with prefix `p` are boolean read-only functions (predicates) that compare an incoming message (`m`) with respect to the local state (`ls`); for example, `pd` can return true if `m<s`. The first pattern is a *discard* pattern where the message is simply discarded if `pd` is true. This pattern is prevalent in distributed systems with votes/versions; old votes/versions tend to be discarded (*e.g.*, our example in §3.1). The *increment* pattern performs an increment-by-one update if `pi` is true, which is also quite common in many protocols (*e.g.*, counting commit acknowledgements). The *constant* pattern changes the local state to a constant whenever `pc` is true. Finally, the *modify* pattern changes the local state whenever `pm` is true.

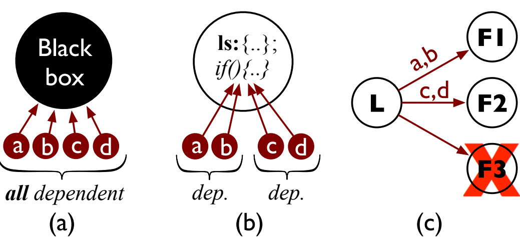


Figure 4: **LMI and CMI.** *The figures illustrate (a) a black-box approach, (b) local-message independence with white-box knowledge, and (c) crash-message independence.*

Based on these patterns, we can apply LMI in the following ways. (1) `m1` is independent of `m2` if `pd` is true on any of `m1` and `m2`. That is, if `m1` (or `m2`) will be discarded, then it does not need to be re-ordered with other messages. (2) `m1` is independent of `m2` if `pi` (or `pc`) is true on both `m1` and `m2`. That is, the re-orderings do not matter because the local state is monotonically increasing by one (or changed to the same constant). (3) `m1` and `m2` are dependent if `pm` is true on `m1` and `pd` is not true on `m2`. That is, since both messages modify the local state in unique ways, then the re-orderings can be "interesting" and hence should be exercised. All these rules are continuously evaluated before every event is enabled. If multiple cases are true, dependency has higher precedence than independency.

Overall, LMI allows dmck to smartly skip redundant re-orderings by leveraging simple patterns. The job of the tester is to find the message processing patterns from a target protocol and write *protocol-specific rules* (*i.e.*, filling in the content of the four LMI predicate functions (`pd`, `pi`, `pc`, and `pm`) specific to the target protocol). As an example, for our simple leader election protocol (§3.1), `pd` can be as simple as: return `m.vote < ls.myVote`.

3.3.2 Crash-Message Independence (CMI)

Figure 4c illustrates the motivation behind our next policy. The figure resembles an atomic broadcast protocol where a leader (`L`) sends commit messages to the followers (`F`s). Let's assume commit messages `ab` to `F1` and `cd` to `F2` are still in flight (*i.e.*, currently outstanding in the dmck; not shown). In addition, the dmck would like to crash `F3`, which we label as a crash event `X`. The question we raise is: how should `X` be re-ordered with respect to other outstanding messages (`a`, `b`, `c`, and `d`)?

As we mentioned before, we find *no* single dmck that incorporates crash semantics into reduction policies. As an implication, in our example, the dmck must reorder `X` with respect to other outstanding messages, generating executions `Xabcd`, `aXbcd`, `abXcd`, and so on. Worse, when `abcd` are reordered, `X` will be reordered again. We find this as one major fundamental problem why existing dmcks do not scale with the inclusion of failures.

To solve this, we introduce crash-message independence (CMI) which defines *independency relationship between a to-be-injected crash and outstanding messages*. The key lies in these two crash reaction patterns (global vs. local impact) running on the surviving nodes (*e.g.*, the leader node in Figure 4c).

```
Global impact:            Local impact:
if (pg(X,ls))              if (pl(X,ls))
  modify(ls);                modify(ls);
  sendMsg();
```

The functions with prefix `p` are predicate functions that compare the crash event `X` with respect to the surviving node's local state (*e.g.*, the leader's local state). The `pg` predicate in the *global-impact* pattern defines that the crash `X` during the local state `ls` will lead to a local state change *and* new outgoing messages (*e.g.*, to other surviving nodes). Here, no reduction can be done because the new crash-induced outgoing messages must be re-ordered with the current outstanding messages. On the other hand, reduction opportunities exist within the *local-impact* pattern, wherein the `pl` predicate specifies that the crash will just lead to a local state change but not new messages, which implies that the crash does not need to be re-ordered.

Based on the two crash impact patterns, we apply CMI in the following ways. Given a local state `ls` at node `N`, a peer failure `X`, and outstanding messages ($m_1...m_n$) from `N` to other surviving peers, CMI performs: (1) If `pl` is true, then `X` and $m_1...m_n$ are independent. (2) If `pg` is true, then `X` and $m_1...m_n$ are dependent. In Figure 4c for example, if `pl` is true in node `L`, then `X` does not impact outstanding messages to `F1` and `F2`, and thus `X` is independent to `abcd`; exercising `Xabcd` is sufficient.

An example of CMI application is a quorum-based write protocol. If a follower crash occurs and quorum is still established, the leader will just decrease the number of followers (local state change only). Here, for the protocol-specific rules, the tester can specify `pl` with `#follower >= majority` and `pg` with the reverse. Overall, CMI helps dmck scale with the inclusion of failures, specifically by skipping redundant re-orderings of crashes with respect to outstanding messages.

3.3.3 Crash Recovery Symmetry (CRS)

Before we discuss our next reduction policy, we emphasize again the difference between message event and crash/reboot event. Message events are generated by the target system, and thus dmck can only reduce the number of re-orderings (but it cannot reduce the events). Contrary, crash events are generated by dmck, and thus there exists opportunities to reduce the number of injected crashes. For example, in Figure 4c, in addition to crashing `F3`, the dmck can also crash `F1` and `F2` in different executions, but that might not be necessary.

To omit redundant crashes, we develop crash recovery symmetry (CRS). The intuition is that some crashes often lead to symmetrical recovery behaviors. For example, let's assume a 4-node system with node roles `FFFL`. At this state, crashing the first or second or third node perhaps lead to the same recovery since all of them are followers, and thereby injecting one follower crash could be enough. Further on, if the system enters a slightly different state, `FFLF`, crashing any of the followers might give the same result as above. However, crashing the leader in the two cases (`N4` in the first case and `N3` in the second) should perhaps be treated differently because the recovery might involve the dead leader ID. The goal of CRS is to help dmck with crash decision.

The main question in implementing CRS is: how to incorporate crash recovery semantics into dmck? Our solution is to compute *recovery abstract global state* (`rags`), a simple and concise representation of crash recovery. CRS builds `rags` with the following steps:

First, we define that two recovery actions are symmetrical if they produce the same messages and change the same local states in the same way.

Second, we extract recovery logic from the code by flattening the predicate-recovery pairs (*i.e.*, recovery-related `if` blocks). Figure 5 shows a simple example. Different recovery actions will be triggered based on which recovery predicate (pr_1, pr_2, or pr_3) is true. Each predicate depends on the local state and the information about the crashing node. Our key here is to map each predicate-recovery pair to this formal pattern:

```
if (pr_i(ls, C.ls))
   modify(rals_i);
   (and/or)
   sendMsg(rals_i);
```

Here, pr_i is the recovery predicate for the i-th recovery action, and $rals_i$ is the recovery abstract local state (*i.e.*, a subset of all fields of the local state involved in recovery). That is, each recovery predicate defines what recovery abstract local state that matters (*i.e.*, $pr_i \rightarrow \{rals_i\}$). For example, in Figure 5, if pr_1 is true, then $rals_1$ only contains the `follower` variable; if pr_3 is true, $rals_3$ contains `role` and `leaderId` variables.

Third, before we crash a node, we check which pr_i will be true on each surviving node and then obtain the $rals_i$. Next, we combine $rals_i$ of all surviving nodes and *sort* them into a recovery abstract global state (`rags`); sorting `rags` helps us exploit topological symmetry (*e.g.*, individual node IDs often do not matter).

Fourth, given a plan to crash a node, the algorithm above gives us the `rags` that represents the corresponding recovery action. We also maintain a history of `rags`

```
broadcast()  sendMsgToAll(role, leaderId);
quorumOkay()  return (follower > nodes / 2);

// pr1
if (role == L && C.role == F && quorumOkay())
  follower--;
// pr2
if (role == L && C.role == F && !quorumOkay())
  follower = 0;
  role = S;
  broadcast();
// pr3
if (role == F && C.role == L)
  leaderId = myId;
  broadcast();
```

Figure 5: **Crash Recovery in Leader Election.** *The figure shows a simplified example of crash recovery in a leader election protocol. The code runs in every node.* `C` *implies the crashing node; each node typically has a view of the states of its peers. Three predicate-recovery pairs are shown* ($\texttt{pr}_1$*,* $\texttt{pr}_2$*, and* $\texttt{pr}_3$*). In the first, if quorum still exists, the leader simply decrements the follower count. In the second, if quorum breaks, the leader falls back to searching mode* (`S`)*. In the third, if the leader crashes, the node (as a follower) votes for itself and broadcasts the vote to elect a new leader.*

of previous injected crashes. If the `rags` already exists in the history, then the crash is skipped because it will lead to a symmetrical recovery of the past.

To recap with a concrete example, let's go back to the case of `FFFL` where we plan to enable crash(`N1`). Based on the code in Figure 5, the `rags` is {*, ∅, ∅, `#follower=3`}; * implies the crashing node, ∅ means there is no true predicate at the other two follower nodes, and `#follower=3` comes from $\texttt{rals}_1$ of $\texttt{pr}_1$ of `N4` (the leader). CRS will sort this and check the history, and assuming no hit, then crash(`N1`) will be enabled. In another execution, SAMC finds that crash(`N2`) at `FFFL` will lead to `rags`:{∅, *, ∅, `#follower=3`}, which after sorting will hit the history, and hence crash(`N2`) is skipped. If the system enters a different state `FFLF`, no follower crash will be injected, because the `rags` will be the same as above. In terms of leader crash, crashing the leader in the two cases will be treated differently because in a leader crash, $\texttt{pr}_3$ is true on followers and $\texttt{pr}_3$ involves `leaderId` which is different in the two cases.

In summary, the foundation of CRS is the computation of recovery abstract global state (`rags`) from the crash recovery logic extracted from the target system via the $\texttt{pr}_i \rightarrow \{\texttt{rals}_i\}$ pattern. We believe this extraction method is simple because CRS does *not* need to know the specifics of crash recovery; CRS just needs to know what variables are involved in recovery (*i.e.*, the `rals`) .

3.3.4 Reboot Synchronization Symmetry (RSS)

Reboots are also essential to exercise (§2.3), but if not done carefully, will introduce more scalability problems. Reboot reduction policy is needed to help dmck inject reboots "smartly". The intuition behind reboot synchronization symmetry (RSS) is similar to that of CRS. When a node reboots, it typically *synchronizes* itself with the peers. However, a reboot will not lead to a new scenario if the current state of the system is similar to the state when the node crashed. To implement RSS, we extract reboot-synchronization predicates and the corresponding actions. Since the overall approach is similar to CRS, we omit further details.

In our experience RSS is extremely powerful. For example, it allows us to find deep bugs involving multiple reboots in the ZooKeeper atomic broadcast (ZAB) protocol. RSS works efficiently here because reboots in ZAB are only interesting if the live nodes have seen new commits (*i.e.*, the dead node falls behind). In contrast, a black-box dmck without RSS initiates reboots even when the live nodes are in similar states as in before the crash, prolonging the discovery of deep bugs.

3.4 Pattern Extraction

We have presented four general, simple, and powerful semantic-aware reduction policies along with the generic event processing patterns. With this, testers can write protocol-specific rules by extracting the patterns from their target systems. Given the patterns described in previous sections, a tester must perform what we call as "extraction" phase. Here, the tester must extract the patterns from the target system and write protocol-specific rules specifically by filling in the predicates and abstractions as defined in previous sections; in Section 4.2, we will show a real extraction result (*i.e.*, real rules). Currently, the extraction phase is manual; we leave automated approaches as a future work (§6). Nevertheless, we believe manual extraction is bearable for several reasons. First, today is the era of DevOps [36] where developers are testers and vice versa; testers know the internals of their target systems. This is also largely true in cloud system development. Second, the processing patterns only cover high-level semantics; testers just fill in the predicates and abstractions but no more details. In fact, simple semantics are enough to significantly help dmck go faster to deeper states.

4 Implementation and Integration

In this section, we first describe our SAMC prototype, SAMPRO, which we built from scratch because existing

Local-Message Independence (LMI)	Crash-Message Independence (CMI)	Crash Recovery Symmetry (CRS)	RSS

bool pd : !newVote(m, s);
 	
bool pm : newVote(m, s);
 	
bool newVote(m, s) :
 if (m.ep > s.ep)
  ret 1;
 else if (m.ep == s.ep)
  if (m.tx > s.tx)
   ret 1;
  else if (m.tx == s.tx &&
         m.lid > s.lid)
   ret 1;
 	
 ret 0;
	bool pg (s, X) :
	 if (s.rl == F && X.rl == L)
	  ret 1;
	 if (s.rl == L && X.rl == F
	     && !quorumAfterX(s)
	  ret 1;
	 if (s.rl == S && X.rl == S)
	  ret 1;
	
	bool pl (s, X) :
	 if (s.rl == L && X.rl == F
	     && quorumAfterX(s))
	  ret 1;
	
	bool quorumAfterX(s) :
	  ret ((s.fol-1) >=
	       s.all/2);
		bool pr1(s,C):
		 if (s.rl == L && C.rl == F
		     && quorumAfterX(s))
		  ret 1;
		rals1:{rl,fol,all};
		
		bool pr2(s,C):
		 if (s.rl == L && C.rl == F
		     && !quorumAfterX(s))
		  ret 1;
		rals2: {rl,fol,lid,ep,tx,clk}
		
		bool pr3(s,C):
		 if (s.rl == F && c.rl == L)
		  ret 1;
		rals3: {rl,fol,lid,ep,tx,clk}
		
		bool pr4:
		 if (s.rl == S)
		  ret 1;
		rals4: {rl,lid,ep,tx,clk}
			(**)
			See
			caption

Table 1: **Protocol-Specific Reduction Rules for ZLE.** *The code above shows the actual protocol-specific rules for ZLE protocol. These rules are the inputs to the four reduction policies. The rule for ZLE's RSS is not shown (it is similar to ZLE's CRS) and many variables are abbreviated (ep: epoch, tx: latest transaction ID, lid: leader ID, rl: role, fol: follower count, all: total node count, clk: logical clock, L: leading, F: following, S: searching, X/C: crashing node). LMI* `pc` *and* `pi` *predicates are not used for ZLE, but used for other protocols.*

dmcks are either proprietary [51] or only work on restricted high-level languages (*e.g.*, Mace [34]). We will then describe SAMPRO integration to three widely popular cloud systems, ZooKeeper [28], Hadoop/Yarn [47], and Cassandra [35]. Prior to SAMPRO, there was no available dmck for these systems; they are still tested via unit tests, and the test code size is bigger than the main code, but the tests are far from reaching deep bugs.

4.1 SAMPRO

SAMPRO is written in 10,886 lines of code in Java, which includes all the components mentioned in Section 2.1 and Figure 1. The detailed anatomy of dmck has been thoroughly explained in literature [22, 25, 34, 43, 51], and therefore for brevity, we will not discuss many engineering details. We will focus on SAMC-related parts.

We design SAMPRO to be highly portable; we do not modify the target code base significantly as we leverage a mature interposition technology, AspectJ, for interposing network messages and timeouts. Our interposition layer also sends local state information to the SAMPRO server. SAMPRO is also equipped with crash and reboot scripts specific to the target systems. The tester can specify a budget of the maximum number of crashes and reboots to inject per execution. SAMPRO employs basic reduction mechanisms and advanced reduction policies as described before. We deploy safety checks at the server (*e.g.*, no two leaders). If a check is violated, the trace that led to the bug is reported and can be deterministically replayed in SAMPRO. Overall, we have built all the necessary features to show the case of SAMC. Other features such as intra-node thread interleavings [25], scale-out parallelism [44], and virtual clock for network delay [51] can be integrated to SAMPRO as well.

4.2 Integration to Target Systems

In our work, the target systems are ZooKeeper, Hadoop 2.0/Yarn, and Cassandra. ZooKeeper [28] is a distributed synchronization service acting as a backbone of many distributed systems such as HBase and High-Availability HDFS. Hadoop 2.0/Yarn [47] is the current generation of Hadoop that separates cluster management and processing components. Cassandra [35] is a distributed key-value store derived from Amazon Dynamo [13].

In total, we have model checked 7 protocols: ZooKeeper leader election (ZLE) and atomic broadcast (ZAB), Hadoop cluster management (CM) and speculative execution (SE), and Cassandra read/write (RW), hinted handoff (HH) and gossiper (GS). These protocols are highly asynchronous and thus susceptible to message re-orderings and failures.

Table 1 shows a real sample of protocol-specific rules that we wrote. Rules are in general very short; we only wrote 35 lines/protocol on average. This shows the simplicity of SAMC's integration to a wide variety of distributed system protocols.

5 Evaluation

We now evaluate SAMC by presenting experimental results that answer the following questions: (1) How fast is SAMC in finding deep bugs compared to other state-of-the-art techniques? (2) Can SAMC find new deep bugs? (3) How much reduction ratio does SAMC provide?

To answer the first question, we show SAMC's effectiveness in finding old bugs. For this, we have integrated SAMPRO to old versions of our target systems that carry deep bugs: ZooKeeper v3.1.0, v3.3.2, v3.4.3, and v3.4.5, Hadoop v2.0.3 and v2.2.0, and Cassandra v1.0.1 and v1.0.6. To answer the second question, we have integrated SAMPRO to two recent stable versions: ZooKeeper v3.4.6 (released March 2014) and Hadoop v2.4.0 (released April 2014). In total, we have integrated SAMPRO to 10 versions, showing the high portability of our prototype. Overall, our extensive evaluation exercised more than 100,000 executions and used approximately 48 full machine days.

5.1 Speed in Finding Old Bugs

This section evaluates the speed of SAMC vs. state-of-the-art techniques in finding old deep bugs. In total, we have reproduced 12 old deep bugs (7 in ZooKeeper, 3 in Hadoop, and 2 in Cassandra). Figure 6 illustrates the complexity of the deep bugs that we reproduced.

Table 2 shows the result of our comparison. We compare SAMC with basic techniques (DFS and Random) and advanced state-of-the-art techniques such as black-box DPOR ("bDP") and Random+bDP ("rDP"). Black-box DPOR is the MODIST-style of DPOR (§2.2). We include Random+DPOR to mimic the way MODIST authors found bugs faster (§2.4). The table shows the number of executions to hit the bug. As a note, software model checking with the inclusion of failures takes time (back-and-forth communications between the target system and the dmck server, killing and restarting system processes multiple times, restarting the whole system from a clean state, etc.). On average, each execution runs for 40 seconds and involves a long sequence of 20-120 events including the necessary crashes and reboots to hit the bug. We do not show the result of running DFS because it never hits most of the bugs.

Based on the result in Table 2, we make several conclusions. First, with SAMC, we prove that smart systematic approaches can reach to deep bugs quickly. We do not need to revert to randomness or incorporate checkpoints. As a note, we are able to reproduce every deep bug that we picked; we did not skip any of them. (Hunting more deep bugs is possible, if needed).

MapReduce-5505: (1) A job finishes, (2) Application manager (AM) sends a "remove-app" message to Resource Manager (RM), (3) RM receives the message, (4) AM is unregistering, (5) *RM crashes* before completely processes the message, (6) AM finishes unregistering, (7) *RM reboots* and reads the old state file, (8) RM thinks the job has never started and runs the job again.

Cassandra-3395 (1) Three nodes N1-3 started and formed a ring, (2) Client writes data, (3) *N3 crashes*, (4) Client updates the data via N1; N3 misses the update, (5) *N3 reboots*, (6) N1 begins the hinted handoff process, (7) Another client reads the data with strong consistency via N1 as a coordinator, (8) N1 and N2 provide the updated value, but N3 still provides the stale value, (9) The coordinator gets "confused" and returns the stale value to the client!

Figure 6: **Complexity of Deep Bugs.** *Above are two sample deep bugs in Hadoop and Cassandra. A sample for ZooKeeper was shown in the introduction (§1). Deep bugs are complex to reproduce; crash and reboot events must happen in a specific order within a long sequence of events (there are more events behind the events we show in the bug descriptions above). To see the high degree of complexity of other old bugs that we reproduced, interested readers can click the issue numbers (hyperlinks) in Table 2.*

Second, SAMC is one to two orders of magnitude faster compared to state-of-the-art techniques. Our speed-up is up to 271x (33x on average). But most importantly, there are bugs that other techniques cannot find even after 5000 executions (around 2 days). Here, SAMC's speed-up factor is potentially much higher (labeled with "⇑"). Again, in the context of dmck (a process of hours/days), large speed-ups matter. In many cases, state-of-the-art policies such as bDP and rDP cannot reach the bugs even after very long executions. The reasons are the two problems we mentioned earlier (§2.4). Our micro-analysis (not shown) confirmed our hypothesis that non-SAMC policies frequently make redundant crash/reboot injections and event re-orderings that anyway lead to insignificant state changes.

Third, Random is truly "random". Although many previous dmcks embrace randomness in finding bugs [34, 51], when it comes to failure-induced bugs, we have a different experience. Sometimes Random is as competitive as SAMC (*e.g.*, ZK-975), but sometimes Random is much slower (*e.g.*, ZK-1419), or worse Random sometimes did not hit the bug (*e.g.*, ZK-1492, MR-5505). We find that some bugs require crashes

Issue#	Protocol	E	C	R	#Executions bDP	RND	rDP	SAMC	Speed-up of SAMC vs. bDP	RND	rDP
ZooKeeper-335	ZAB	120	3	3	↑5000	1057	↑5000	117	⇑43	9	⇑43
ZooKeeper-790	ZLE	21	1	1	14	225	82	7	2	32	12
ZooKeeper-975	ZLE	21	1	1	967	71	163	53	18	1	3
ZooKeeper-1075	ZLE	25	3	2	1081	86	250	16	68	5	16
ZooKeeper-1419	ZLE	25	3	2	924	2514	987	100	9	25	10
ZooKeeper-1492	ZLE	31	1	0	↑5000	↑5000	↑5000	576	⇑9	⇑9	⇑9
ZooKeeper-1653	ZAB	60	1	1	945	3756	3462	11	86	341	315
MapReduce-4748	SE	25	1	0	22	6	6	4	6	2	2
MapReduce-5489	CM	20	2	1	↑5000	↑5000	↑5000	53	⇑94	⇑94	⇑94
MapReduce-5505	CM	40	1	1	1212	↑5000	1210	40	30	⇑125	30
Cassandra-3395	RW+HH	25	1	1	2552	191	550	104	25	2	5
Cassandra-3626	GS	15	2	1	↑5000	↑5000	↑5000	96	⇑52	⇑52	⇑52

Table 2: **SAMC Speed in Finding Old Bugs.** *The first column shows old bug numbers in ZooKeeper, Hadoop, and Cassandra that we reproduced. The bug numbers are clickable (contain hyperlinks). The protocol column lists where the deep bugs were found; full protocol names are in §4.2. "E", "C" and "R" represent the number of events, crashes, and reboots necessary to hit the bug. The numbers in the middle four columns represent the number of executions to hit the bug across different policies. "bDP", "RND", and "rDP" stand for black-box DPOR (in* MODIST*), random, and random + black-box DPOR respectively. The SAMC column represents our reduction policies and rules. The last three columns represent the speed-ups of SAMC over the other three techniques. We stop at 5000 executions (around 2 days) if the bug cannot be found; potentially many more executions are required to hit the bug (labeled with "↑"). Thus, speed-up numbers marked with "⇑" are potentially much higher. In the experiments above, the bugs are reproduced using 3-4 nodes. We also have run DFS but do not show the result because in most cases DFS cannot hit the bugs. For model checking the SE protocol, "1C" means one straggler; we emulate a node slowdown as a failure event by modifying the progress report of the "slow" node. SE involves 20+ events but most of them are synchronized stages and cannot be re-ordered, which explains why the SE bug can be found quickly with all policies.*

and/or reboots to happen at very specific points, which is probabilistically hard to reach with randomness. With SAMC, we show that being systematic and semantic aware is consistently effective.

5.2 Ability of Finding New Bugs

The previous section was our main focus of evaluation. In addition to this, we have integrated SAMPRO to recent stable versions of ZooKeeper (v3.4.6, released March 2014) and Hadoop (v2.4.0, released April 2014). In just hours of deployment, we found 1 new ZLE bug involving 2 crashes, 2 reboots, and 52 events, and 1 new Hadoop speculative execution bug involving 2 failures and 32 events. These two new bugs are distributed data race bugs. The ZLE bug causes the ZooKeeper cluster to create two leaders at the same time. The Hadoop bug causes a speculative attempt on a job that is wrongly moved to a scheduled state, which then leads to an exception and a failed job. We can deterministically reproduce the bugs multiple times and we have reported the bugs to the developers. Currently, the bugs are still marked as major and critical, the status is still open, and the resolution is still unresolved.

We also note that in order to unearth more bugs, a dmck must have several complete features: workload generators that cover many protocols, sophisticated perturbations (*e.g.*, message re-ordering, fault injections) and detailed checks of specification violations. Further discussions can be found in our previous work [23]. Currently, SAMPRO focuses on speeding up the perturbation part. By deploying more workload generators and specification checks in SAMPRO, more deep bugs are likely to be found. As an illustration, the 94 deep bugs we mentioned in Section 2.3 originated from various protocols and violated a wide range of specifications.

5.3 Reduction Ratio

Table 3 compares the reduction ratio of SAMC over black-box DPOR (bDP) with different budgets (#crashes and #reboots). This evaluation is slightly different than the bug-finding speed evaluation in Section 5.1. Here, we measure how many executions in bDP are considered redundant based on our reduction policies and protocol-specific rules. Specifically, we run bDP for 3000 executions and run SAMC policies on the side to mark the redundant executions. The reduction ratio is then 3000 divided by the number of non-redundant executions. Table 3 shows that SAMC provides between 37x-166x execution reduction ratio in model checking ZLE and ZAB protocols across different crash/reboot budgets.

Table 3b shows that with each policy the execution reduction ratio increases when the number of crashes and

C	R	Execution Reduction Ratio in ZLE	ZAB
1	1	37	93
2	2	63	107
3	3	103	166

C	R	Execution Reduction Ratio in ZLE with All	LMI	CMI	CRS	RSS
1	1	37	18	5	4	3
2	2	63	35	6	5	5
3	3	103	37	9	9	14

Table 3: **SAMC Reduction Ratio.** *The first table shows the execution reduction ratio of SAMC over black-box DPOR (bDP) in checking ZLE and ZAB under different crash/reboot budgets. "C" and "R" are the number of crashes and reboots. The second table shows the execution reduction ratio in ZLE with individiual policies over black-box DPOR (bDP).*

reboots increases. With more crashes and reboots, the ZLE protocol generates more messages and most of them are independent, and thus the LMI policy has more opportunities to remove redundant message re-orderings. Similary, the crash and reboot symmetry policies give better benefits with more crashes and reboots. The table also shows that LMI provides the most reduction. This is because the number of message events is higher than crash and reboot events (as also depicted in Table 2).

We now discuss our reduction ratio with that of DIR [25]. As discussed earlier (§2.2), DIR records local exploration (thread interleavings) and replays future incoming messages whenever possible, reducing the work of global exploration. If the target system does not have lots of thread interleavings, DIR's reduction ratio is estimated to be between 10^1 to 10^3 (§5 of [25]). As we described earlier (§2.2), DIR is orthogonal to SAMC. Thus, the reduction ratios of SAMC and DIR are complementary; when both methods are combined, there is a potential for a higher reduction ratio. The DIR authors also hinted that domain knowledge can guide dmcks (and also help their work) to both scale and hit deep bugs (§8 of [25]). SAMC has successfully addressed such need.

Finally, we note that in evaluating SAMC, we use execution reduction ratio as a primary metric. Another classical metric to evaluate a model checker is state coverage (*e.g.*, a dmck that covers more states can be considered a more powerful dmck). However, in our observation state coverage is not a proper metric for evaluating optimization heuristics such as SAMC policies. For example, if there are three nodes ABC that have the same role (*e.g.*, follower), a naive black-box dmck will crash each node and covers three distinct states: *BC, A*C and AB*. However, with a semantic-aware approach (*e.g.*, symmetry), we know that covering one of the states is sufficient. Thus, less state coverage does not necessarily imply a less powerful dmck.

6 Discussion

In this section, we discuss SAMC's simplicity, generality and soundness. We would like to emphasize that the main goal of this paper is to show the power of SAMC in finding deep bugs both quickly and systematically, and thus we intentionally leave some subtasks (*e.g.*, automated extraction, soundness proofs) for future work.

6.1 Simplicity

In previous sections, we mentioned that policies can be written in few lines of code. Besides LOC, simplicity can be measured by how much time is required to understand a protocol implementation, extract the patterns and write the policies. This time metric is unfortunately hard to quantify. In our experience, the bulk of our time was spent in developing SAMPRO from scratch and integrating policies to dmck mechanisms (§2.1). However, the process of understanding protocols and crafting policies requires a small effort (*e.g.*, few days per protocol to the point where we feel the policies are robust). We believe that the actual developers will be able to perform this process much faster than we did as they already have deeper understandings of their code.

6.2 Generality

Our policies contain patterns that are common in distributed systems. One natural question to ask is: how much semantic knowledge should we expose to dmck? The ideal case is to expose as much information as possible as long as it is sound. Since proving soundness and extracting patterns automatically are our future work, in this paper we only propose exposing high-level processing semantics. With advanced program analysis tools that can analyze deep program logic, we believe more semantic knowledge can be exposed to dmck in a sound manner. For example, LMI can be extended to include commutative modifications. This is possible if the program analysis can verify that the individual modification does not lead to other state changes. This will perhaps be the point where symbolic execution and dmck blend in the future (§7).

Nevertheless, we find that high-level semantics are powerful enough. Beyond the three cloud systems we target in this paper, we have been integrating SAMC to MaceMC [34]. MACEMC already employs random exploration policies to model check Mace-based distributed systems such as Mace-based Chord and Pastry. To integrate SAMC, we first must re-implement DPOR in

MaceMC (existing DPOR implementation in MaceMC is proprietary [25]). Then, we have written 18 lines of LMI protocol-specific rules for Chord and attain two orders of magnitude of reduction in execution. This shows the generality of SAMC to many distributed systems.

6.3 Soundness

SAMC policies only skip re-orderings and crash/reboot events that are redundant by definition, however currently our version of SAMC is not sound; the unsound phase is the manual extraction process. For example, if the tester writes a wrong predicate definition (*e.g.*, `pd`) that is inconsistent with what the target system defines, then soundness (and correctness) is broken. Advanced program analysis tools can be developed to automate and verify this extraction process and make SAMC sound. Currently, the fact that protocol-specific rules tend to be short might also help in reducing human errors. Our prototype, SAMPRO, is no different than other testing/verification tools; full correctness requires that such tools to be free of bugs and complete in checking all specifications, which can be hard to achieve. Nevertheless, we want to bring up again the discussion in Section 2.4 that dmck's scalability and ability to find deep bugs in complex distributed systems are sometimes more important than soundness. We leave soundness proofs for future work, but we view this as a small limitation, mainly because we have successfully shown the power of SAMC.

7 Related Work

We now briefly discuss more related work on dmck and other approaches to systems verification and testing.

Formal model checking foundations such as partial order reduction [17, 20], symmetry [9, 45], and abstraction [10], were established more than a decade ago. Here, classical model checkers require system models and mainly focus on state-space reduction. Implementation-level model checkers on the other hand are expected to find real bugs in addition to being efficient.

Symbolic execution is another powerful formal method to verify systems correctness. Symbolic execution also faces an explosion problem, specifically the path explosion problem. A huge body of work has successfully addressed the problem and made symbolic execution scale to large (non-distributed) software systems [3, 6, 8, 11, 55]. Symbolic execution and model checking can formally be combined into a more powerful method [4], however this concept has not permeated the world of distributed systems; it is challenging to track symbolic values across distributed nodes.

Reliability bugs are often caused by incorrect handling of failures [23, 24]. Fault-injection testing however is challenging due to the large number of possible failures to inject. This challenge led to the development of efficient fault-injection testing frameworks. For example, AFEX [1] and LFI [39] automatically prioritize "high-impact targets" (*e.g.*, unchecked system calls). These novel frameworks target non-distributed systems and thus the techniques are different than ours.

Similarly, recent work highlights the importance of testing faults in cloud systems (*e.g.*, FATE [23], SETSUDO [30], PREFAIL [31], and OpenStack fault-injector [32]). As mentioned before (§2.2), these frameworks are not a dmck; they cannot re-order concurrent messages and failures and therefore cannot catch distributed concurrency bugs systematically.

The threat of multiple failures to systems reliability already existed since the P2P era; P2P systems are susceptible to "churn", the continuous process of node joining and departing [42]. Many dmcks such as MACEMC [34] and CrystalBall [50] evaluate their approaches on P2P systems. Interestingly, we find that they mainly re-order join messages. To our understanding, based on their publications, they did not inject and control node departures. CrystalBall authors mentioned about running churns, but only as part of their workloads, not as events that the dmck can re-order. This illustrates the non-triviality of incorporating failures in dmck.

The deep bugs we presented can be considered as concurrency bugs (in distributed nature). For non-distributed systems, there has been an abundance of innovations in detecting, avoiding, and recovering from concurrency bugs [29, 33, 38, 48]. They mainly target threads. For dmck, we believe more advancements are needed to unearth distributed concurrency bugs that still linger in cloud systems.

Finally, the journey in increasing cloud dependability is ongoing; cloud systems face other issues such as bad error handling code [54], performance failures [14], corruptions [15], and many others. Exacerbating the problem, cloud systems are becoming larger and geographically distributed [37, 46, 56]. We believe cloud systems will observe more failures and message re-orderings, and therefore our work and future dmck advancements with the inclusion of failures will play an important role in increasing the reliability of future cloud systems.

8 Conclusion

Cloud systems face complex failures and deep bugs still linger in the cloud. To address present reliability challenges, dmcks must incorporate complex failures, but existing dmcks do not scale in this regard. We strongly be-

lieve that without semantic knowledge dmck hits a scalability wall. In this paper, we show a strong case that the SAMC principle can elegantly address this scalability problem. SAMC is simple and powerful; with simple semantic knowledge, we show that dmcks can scale significantly. We presented four specific reduction policies, but beyond this, we believe our work triggers the discussion of two important research questions: *what* other semantic knowledge can scale dmck and *how* to extract white-box information from the target system? We hope (and believe) that the SAMC principle can trigger future research in this space.

9 Acknowledgments

We thank Bryan Ford, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We would also like to thank Yohanes Surya and Teddy Mantoro for their support. This material is based upon work supported by the NSF (grant Nos. CCF-1321958 and CCF-1336580). The experiments in this paper were performed in the Utah Emulab[1] [49] and NMC PRObE[2] [19] testbeds, supported under NSF grants Nos. CNS-1042537 and CNS-1042543.

References

[1] Radu Banabic and George Candea. Fast black-box testing of system recovery code. In *Proceedings of the 2012 EuroSys Conference (EuroSys)*, 2012.

[2] Ken Birman, Gregory Chockler, and Robbert van Renesse. Towards a Cloud Computing Research Agenda. *ACM SIGACT News*, 40(2):68–80, June 2009.

[3] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proceedings of the 2011 EuroSys Conference (EuroSys)*, 2011.

[4] J. R. Burch, E. M. Clarke, K L. McMillan, D L. Dill, and L J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[5] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems Export. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[1] http://www.emulab.net
[2] http://www.nmc-probe.org

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[9] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *10th International Conference on Computer Aided Verification (CAV)*, 1998.

[10] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 1994.

[11] Heming Cui, Gang Hu Columbia, Jingyue Wu, and Junfeng Yang. Verifying Systems Rules Using Rule-Directed Symbolic Execution. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[14] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[15] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HARDFS: Hardening HDFS with Selective and Lightweight Versioning. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[16] E. Allen Emerson, Somesh Jha, and Doron Peled. Combining Partial Order and Symmetry Reductions. In *3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1997.

[17] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *The 33th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[19] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 38(3), June 2013.

[20] Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. volume 1032, 1996.

[21] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997.

[22] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[23] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[24] Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[25] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[26] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[28] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[29] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated Concurrency-Bug Fixing. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[30] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. SETSUDO : Perturbation-based Testing Framework for Scalable Distributed Systems. In *Conference on Timely Results in Operating Systems (TRIOS)*, 2013.

[31] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[32] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. On Fault Resilience of OpenStack. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[33] Baris Kasikci, Cristian Zamfir, and George Candea. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[34] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[35] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

[36] Thomas A. Limoncelli and Doug Hughe. LISA '11 Theme – DevOps: New Challenges, Proven Values. *USENIX ;login: Magazine*, 36(4), August 2011.

[37] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[38] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[39] Paul D. Marinescu, Radu Banabic, and George Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[40] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[41] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[42] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2004.

[43] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV)*, 2010.

[44] Jiri Simsa, Randy Bryant, Garth A. Gibson, and Jason Hickey. Scalable Dynamic Partial Order Reduction. In *the 3rd International Conference on Runtime Verification (RV)*, 2012.

[45] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 2010.

[46] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[47] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[48] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and Surviving Data Races using Complementary Schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[49] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[50] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed. Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[51] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[52] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[53] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[54] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[55] Cristian Zamfir and George Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 2010 EuroSys Conference (EuroSys)*, 2010.

[56] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration

Pedro Fonseca
Max Planck Institute for Software Systems (MPI-SWS)

Rodrigo Rodrigues
NOVA University of Lisbon (CITI/NOVA-LINCS)

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Kernel concurrency bugs are notoriously difficult to find during testing since they are only triggered under certain instruction interleavings. Unfortunately, no tools for systematically subjecting kernel code to concurrency tests have been proposed to date. This gap in tool support may be explained by the challenge of controlling precisely which kernel interleavings are executed without modifying the kernel under test itself. Furthermore, to be practical, prohibitive runtime overheads must be avoided and tools must remain portable as the kernel evolves.

In this paper, we propose SKI, the first tool for the systematic exploration of possible interleavings of kernel code. SKI finds kernel bugs in *unmodified* kernels, and is thus directly applicable to different kernels. To achieve control over kernel interleavings in a portable way, SKI uses an adapted virtual machine monitor that performs an efficient analysis of the kernel execution on a virtual multiprocessor platform. This enables SKI to determine which kernel execution flows are eligible to run, and also to selectively control which flows may proceed. In addition, we detail several essential optimizations that enable SKI to scale to real-world concurrency bugs.

We reliably reproduced previously reported bugs by applying SKI to different versions of the Linux kernel and to the FreeBSD kernel. Our evaluation further shows that SKI was able to discover, in widely used and already heavily tested file systems (e.g., ext4, btrfs), several unknown bugs, some of which pose the risk of data loss.

1 Introduction

In the current multi-core era, kernel developers are under permanent pressure to continually increase the performance of kernels through concurrency. Examples of such efforts include reducing the granularity of locking [59], rewriting subsystems to use parallel algorithms [26], and using non-traditional and optimistic synchronization primitives (such as RCU [52] and lock-free data structures [67]). Unfortunately, previous experience has shown that all these efforts are error-prone and can easily lead to kernel concurrency bugs — bugs that are only exposed by a subset of the possible thread interleavings.

In practice, kernel developers find concurrency bugs mostly through manual code inspection [39, 69] and stress testing [14, 64] (i.e., applying intense workloads to increase the chances of triggering concurrency bugs). While useful, both approaches have significant shortcomings: code inspection is labor-intensive and requires significant skill and experience, and stress testing, despite having low overhead and being amenable to automation, offers no guarantees and can easily fail to uncover difficult to find concurrency bugs — i.e., edge cases that are only triggered by a tiny subset of the interleavings. It thus stands to reason that kernel developers could benefit from tools without these limitations.

To this end, we propose a *complementary* testing approach for automatically finding kernel concurrency bugs. Our approach explores the kernel interleaving space in a systematic way by taking full control over the kernel thread interleavings. Similar approaches have been explored for user-mode applications, yielding good results [20, 53, 54], but have not yet been applied to commodity kernels because achieving control over the thread interleavings of kernels involves several challenges. First, to be practical, a concurrency testing tool must be generally applicable, rather than being specific to a particular kernel or kernel version, which precludes kernel-specific modifications. Second, the kernel is the software layer that implements its own thread scheduler, as well as the thread abstraction itself, making the external control of thread interleavings non-trivial. Finally, to

be effective, such a tool must be able to control kernel interleavings while introducing a low overhead.

In this paper, we report on the design and an evaluation of SKI[1], the first tool for the systematic exploration of kernel interleavings to overcome these challenges. To achieve control over kernel interleavings in a portable way, SKI uses an adapted virtual machine monitor that (1) determines the status of the various threads of execution, in terms of being blocked or ready to run, to understand the scheduling restrictions, and (2) selectively blocks a subset of these threads in order to enforce the desired schedule. Notably, these key tasks are achieved without any modification to the kernel and without specific knowledge of the semantics of the kernel's internal synchronization primitives. Furthermore, we propose several optimizations, both at the algorithmic and at the implementation levels, that we found to be important for scaling SKI to real-world concurrency bugs.

We evaluated SKI by testing several file systems in recent versions of the Linux kernel and we found 11 previously unknown concurrency bugs. Of these, several concurrency bugs can cause serious data loss in important file systems (*ext4* and *btrfs*). We also show how SKI can be used to reproduce concurrency bugs that have been previously reported in two different operating systems (Linux and FreeBSD), and compare SKI's performance against the traditional stress testing approach.

We believe that SKI is an important step towards increased kernel reliability on multicore platforms. Nonetheless, there remains significant room for exploiting domain- and kernel-specific knowledge. For instance, in this paper we propose a scheduling algorithm (for performing the schedule exploration) that is generic in the sense that it makes no assumptions about the kernel under test. However, based on the SKI infrastructure, other kernel-specific scheduling algorithms could be implemented, for example, to restrict the interleavings explored to those that affect specific kernel instructions, such as code that was recently modified. Thus, we believe that SKI can provide benefits even beyond those described in this paper, since it can serve as an experimentation framework for different systematic techniques.

The rest of the paper is organized as follows. Section 2 motivates the need for better kernel testing tools. Section 3 presents the design of SKI. Section 4 proposes several optimizations to make SKI scale to real-world concurrency bugs. Section 5 describes the details of our implementation. Section 6 evaluates SKI and Section 7 discusses some of its limitations. In Section 8 we discuss related work and finally we conclude in Section 9.

[1] Systematic Kernel Interleaving explorer

2 Systematic testing

A *systematic* exploration of the interleaving space, in contrast with a stress testing approach, relies on judiciously controlling the thread schedule for each execution of the software under test to maximize the coverage of the interleaving space.

At a high level, systematic approaches increase the effectiveness of testing by avoiding redundant interleavings and prioritizing interleavings that are more likely to expose bugs, e.g., those that differ more from interleavings that have already been explored. It has been shown both analytically and empirically that such methods offer better results than traditional *ad hoc* approaches [20].

To achieve this level of control over interleavings, systematic approaches rely on a custom thread scheduler that implements two basic mechanisms. The first mechanism infers thread liveness to understand which schedules it can choose, which can be achieved by intercepting and understanding the semantics of the synchronization functions. The second mechanism overrides the regular scheduler by allowing only a specifically chosen thread to make progress at any point in time.

In the case of user-mode applications, both of these two essential mechanisms can be easily implemented in a proxy layer (e.g., through *LD_PRELOAD* or *ptrace*) by intercepting all relevant synchronization primitives to infer and override the liveness state of each thread [20, 53, 54]. Unfortunately, a direct application of the user-mode method to the kernel would require modifying the kernel itself, which would suffer from several disadvantages:

- **Lack of portability and API instability.** Any dependency on kernel-internal APIs would a priori limit the portability of the envisioned testing tool, preventing its seamless application across different kernels and even across different versions of the same kernel. In contrast to well-documented, standardized user-space interfaces (e.g., the pthreads API), the internal API of most kernels is not guaranteed to be stable, and in fact typically changes from version to version. In particular, given the current trend towards increased hardware parallelism, kernel synchronization has generally been an active area of development in Linux and other kernels [26, 52].
- **Complexity of the internal interface.** An additional problem with the internal API of the kernel, also noted in previous work [32], is that the semantics of in-kernel synchronization operations are particularly complex. Furthermore, the exact semantics of such operations tend to differ from kernel to

kernel. This calls for solutions that do not require a detailed understanding of these semantics.

- **Other forms of concurrency.** Interrupts are pervasive and critical to kernel code. However, exercising fine-level control over their timing from within the kernel itself would be particularly challenging, as interrupts are scheduled by the hardware.[2]

- **Intrusive testing.** Requiring modifications to the tested software goes against the principles of testing [43] — testing *modified* versions of the software can potentially introduce or elide bugs.

In the next section we explain how SKI overcomes these challenges while enabling the systematic exploration of kernel thread interleavings.

3 SKI: Exploring kernel interleavings

This section presents the design of SKI. We start by providing an overview of our solution (Section 3.1), and then we describe how SKI exercises control over thread interleavings (Section 3.2) and how it gathers the necessary liveness information (Section 3.3). We conclude this section with a description of the scheduling algorithms employed, i.e., the interleavings chosen for each run (Section 3.4).

3.1 Overview

The inputs given to SKI are the initial state of the system under test and the kernel input that is to be tested concurrently (i.e., two or more concurrent system calls). Given these inputs, SKI carries out several test runs corresponding to different concurrent executions, where each test run is fully serialized, i.e., the tool enables only a single thread to execute at each instant. This enables precise control over which interleavings are executed, and allows SKI's scheduler to choose successive runs to improve the interleaving space coverage. Either during or after each test run, a bug detector is used to determine if the test has flagged a possible bug. Such bug detectors can perform simple, generic actions like detecting crashes, or complex, application-specific actions like running a system integrity check after the test run.

As mentioned in the previous section, for SKI's scheduler to gain control over the interleavings executed by the kernel, it must perform two key tasks: inferring thread liveness and overriding the scheduler. To accomplish both without modifying the OS kernel under test, we implement the scheduler of SKI at the level of a modified virtual machine monitor (VMM), taking as input a virtual machine (VM) image that incorporates the initial state of the kernel immediately before the system calls are invoked concurrently. Implementing the scheduler at the VMM level enables it to both observe and control the kernel under test.

This advantage comes, however, at the cost of making it more difficult to implement the two aforementioned key tasks. This is because, at the VMM level, the hypervisor observes a stream of machine instructions to be executed, and has direct access only to the physical resources of the underlying hardware (such as registers or memory contents). These low-level concepts are distant from the abstractions that are implemented by the kernel in software, such as threads and their respective contexts. Furthermore, it would intuitively seem necessary to have access to these abstractions for suspending the execution of a thread and replacing it with another thread.

3.2 Exercising control over threads

To control the progress of threads, SKI relies on the observation that the most widely used kernels (e.g., Linux, Windows, MacOS X, FreeBSD) include a mechanism to allow applications to *pin* threads to individual CPUs (i.e., to specify the thread affinity). This mechanism, provided by kernels to user-mode applications for performance reasons, can be exploited to create a 1:1 mapping between threads (a kernel abstraction) and virtual CPUs (an ISA component, controllable by the VMM). This mapping in turn allows SKI to block and resume a thread execution by simply suspending and resuming the corresponding virtual CPU's execution of machine instructions.

Apart from the user-space threads that invoke system calls, operating systems have another type of threads, which similarly execute kernel code, namely *kernel threads* [7]. Kernel threads are used by the kernel to asynchronously execute tasks. Despite not being associated with user-mode processes, some kernel threads can be pinned to different CPUs from user-space. For kernel threads that cannot be pinned to other CPUs for OS-specific reasons, SKI is not able to explicitly control their schedule and therefore lets the OS schedule them.

To implement the mapping between threads and CPUs, SKI includes, in addition to the modified VMM, a user-mode component that runs inside the VM and issues system calls to pin threads to virtual CPUs (see Sec-

[2] While user-mode signals are similar to interrupts, many programs do not use them and therefore existing user-mode tools do not handle them [20, 53].

tion 5.3). Note that for scalability reasons, each test generally involves only few threads, and hence it suffices to configure a small number of virtual CPUs.

3.3 Inferring liveness

To explore the interleaving space, SKI requires information about whether threads are blocked or able to progress, analogously to what is required by the existing user-mode tools [20, 53, 54]. This requires SKI to be able to identify constructs such as spin-locks or barriers, where a CPU executes a tight loop, constantly checking the value of a memory location for changes. SKI would be impractical if it were not able to detect such constructs, for several reasons. First, executions would take longer because more instructions would be executed (e.g., iterations of a spin loop). Second, because more instructions would be executed, the space of possible interleavings would significantly increase, since the number of possible interleavings is exponential in the length of the test. Third, and most importantly, given the scheduling algorithm that we describe in Section 3.4, two interleavings could be considered different even when they only differ in the number of iterations executed by the polling loop of a spin lock. This would be detrimental to the efficiency of SKI, since many of the explored schedules would be effectively equivalent.

The difficulty in inferring thread liveness is that, from the point of view of the VMM, CPUs are constantly executing instructions. As such, it is difficult to distinguish the normal execution of a program from a polling loop.

One possible solution that we considered, but ultimately rejected, relies on annotating the kernel by specifying the locations within the kernel code where the CPU executes instructions without making any actual progress, namely situations where the kernel is waiting for some event external to the CPU (such as an action performed by some other CPU or a device notification). However, this approach would be laborious, error prone, and non-portable.

Instead, we found several simple heuristics independent of the kernel code that enable the VMM to infer whether a CPU is making progress or not.

H1: Halt heuristic. The first heuristic flags the CPU as non-live when it executes the halt instruction (HLT).[3] According to the instruction set specification, HLT marks the CPU as waiting for interrupts. This instruction is typically used by kernels to implement, in an energy efficient way, the idle thread when the kernel scheduler has no other threads to run. When the CPU subsequently receives an interrupt, it is marked as live again.

H2: Pause heuristic. The second heuristic relies on the observation that kernels use the pause instruction (PAUSE) to efficiently implement spin-locks. In the x86 architecture, the pause instruction has been introduced to avoid wasting bandwidth on the memory bus when a CPU goes into a tight polling loop, and therefore its execution is a good indication that the CPU is spinning on a lock. Thus, when our modified VMM detects the execution of two nearby pause instructions, i.e., within an instruction window of size h_2, it considers the CPU to be non-live and takes note of the memory read-set associated with the instructions executed between the two pause instructions. Pause instructions in close proximity are detected by the VMM by checking, at every pause instruction, whether another pause instruction was recently executed. Later on, when another CPU changes one of the addresses in the read-set, the non-live CPU is optimistically marked as live again.

H3: Loop heuristic. The third heuristic detects situations where the CPU is waiting for some external event, but that are not caught by the second heuristic. This could happen if, for example, a spin-lock were implemented without including the pause instruction. To detect CPUs stuck in a polling loop, our modified VMM maintains a window, of size h_3, of the last few instructions executed by each CPU. If a CPU repeatedly executes the same instructions (i.e., if it executes a loop), and if an instruction in the loop repeatedly reads the same value from the same memory address, the executing context is flagged as non-live after a certain number of loop iterations. Again, SKI takes note of the read-set of detected polling loops to later re-enable the CPU.

H4: Starvation heuristic. As a last resort, in case the above heuristics are not able to detect situations where there is no progress, SKI keeps a count of the number of instructions executed continuously by the current CPU, and, if it exceeds a threshold (h_4), it conservatively presumes that the CPU is no longer making progress. The CPU is marked live again after a certain number of instructions have been executed by the other CPUs. This heuristic ensures the detection, for example, of loops that are missed by H3 if h_3 is set smaller than the loop size.

We determined the values for the thresholds of these heuristics, which remained constant throughout all our tests, through simple experimentation. From our experience, these mechanisms were sufficient to ensure the effectiveness of SKI for a wide range of kernel versions, at both reproducing previously known bugs and at finding unknown bugs.

[3] We focus on the ubiquitous x86 architecture in this paper; the presented ideas, however, can be similarly applied to other architectures.

3.4 Scheduling algorithm

SKI executes a VM multiple times under different schedules to ensure interleaving diversity across the runs. To select and prioritize the interleavings that are to be explored SKI needs to implement a scheduling algorithm.

SKI uses an extension of the PCT algorithm [20], a state-of-the-art algorithm originally developed for user-mode applications, which has been shown to be effective at uncovering user-mode concurrency bugs. SKI extends PCT by supporting interrupts (Section 3.4.2), which is a fundamental requirement for testing operating systems.

Nonetheless, we consider the proposed algorithm to be just one instance from a range of possible algorithms (albeit one that in our experience happens to work well), and developers that make use of the tool might consider adding other, more refined algorithms. For example, it may be possible to develop effective scheduling algorithms that exploit specific characteristics of kernel code.

Since the SKI scheduler must handle both threads and interrupts, it schedules *contexts* instead of threads; we will thus refer to contexts throughout this description.

3.4.1 Background: PCT algorithm

Conceptually the scheduler executes instructions sequentially one by one; that is, at any point during the execution, only one of the live contexts is allowed to progress, and the eligibility of the context to execute another instruction is re-evaluated after each instruction. Through this process, the scheduler is able to effectively control the chosen interleaving. In practice, however, our implementation optimizes this process by using a JIT compiler and by only introducing checks as needed (Section 5).

A strawman design for the scheduler would be to use a fixed ordering of the various contexts, and to run the first context for the longest possible period until it is no longer able to run. At this point, the scheduler chooses the second context to run until either it is also no longer able to run, or the first context becomes able to run again, and so on. While this initial design suffices to create valid schedules and allows tests to finish, it does not create a diverse set of schedules.

To achieve a good diversity of schedules across different runs, the scheduler uses two strategies. The first is to randomly assign initial priorities to the contexts, and use these priorities instead of a fixed order to determine the context that should run at each instant – this is the context with the highest priority among those that are not blocked. The second strategy consists of reducing, at random points during the execution of a test, the priority of the context that is scheduled. If the priority decrease is large enough, this will cause another context to become the one with the highest priority, and therefore this other context will be scheduled to run. By varying both the initial priorities and the location of such *reschedule points* in a controlled way, the scheduler is able to control the range of tested schedules.

The reschedule points are chosen prior to each run by randomly selecting a set of offsets from the start of the test (in terms of the total number of instructions executed) within a certain range. Then, during the execution, whenever the total number of instructions executed reaches one of these offsets, the priority of the currently scheduled context is lowered so that it becomes the lowest-priority context, and thus another runnable context is selected for execution in the next step.

The set of reschedule points is determined according to two parameters: the expected number of execution steps k and the desired number of reschedule points p, with the simple interpretation that there will be up to p reschedules within the first k instructions of the execution of the test (and none thereafter, should the test execute for more than k instructions). That is, for a given k and p, the set of p reschedule points is selected by choosing uniformly at random p offsets from the range $[0,k]$.

3.4.2 Handling interrupts

Given that SKI operates at the level of the virtual machine monitor, it does not have access to the thread abstraction that is used by schedulers for user-mode applications. Thus, instead of scheduling threads, our algorithm schedules CPUs. In addition, another distinction to user-mode schedulers is that the scheduler needs to make decisions regarding when interrupts should be dispatched. Interrupts do not appear in the context of user-mode programs, but we need to control their schedule when testing the kernel for two different reasons. First, concurrency bugs may depend on the interleaving of interrupts, so our algorithm should be able to explore this part of the interleaving space. Second, interrupts are in some cases required for the successful completion of system calls, and therefore interrupts need to be scheduled to conclude the execution of the tests. For example, some system calls are only able to finish if, during their execution, other CPUs handle the TLB flush interrupt.

As the scheduler needs to consider when interrupts are handled, each CPU is tracked as being in one of two different contexts: it may either execute in the context of an interrupt handler (*interrupt-context*), or it may execute outside of the context of any interrupt handlers (*CPU-context*). Each *interrupt-context* is defined by the CPU on which it arrived and by the interrupt number

Schedule 1

```
 CPU1   INT1   CPU2

INST 1
INST 2
INST 3
INST 4
<end>
        INST 1
        INST 2
        <end>
               INST 1
               INST 2
               INST 3
               <end>
```

Schedule 2

```
 CPU1   INT1   CPU2
INST 1
INST 2
<inv>
        INST 1
        INST 2
        <end>
               INST 1
               INST 2
               INST 3
               <end>
INST 3
INST 4
<end>
```

Figure 1: Two examples illustrating schedules produced by SKI. Each schedule involves three contexts, two CPU-contexts and one interrupt-context. Both schedules start with the same initial context priorities. However, Schedule 2 differs from Schedule 1 because it contains one priority inversion (<inv>).

that it represents. From the point of view of the scheduler, *interrupt-contexts* are created, and therefore become schedulable, when the corresponding interrupt arrives on its specific CPU. These execution contexts are, to our scheduler, the equivalent to threads for other systematic exploration algorithms, and as such they need to be detected by the scheduling logic. SKI infers the context by tracking the interrupt handler dispatches and the *IRET* instruction invocations (which are used to return from interrupt handlers). Figure 1 shows examples of schedules involving two *CPU-contexts* and one *interrupt-context*.

To achieve further control over the tests, SKI allows the user to specify a set of execution contexts that are allowed to run during the test. In particular, placing restrictions on the set of eligible execution contexts may be useful in specific testing scenarios, to restrict the scheduling space that is explored.

3.5 Discussion

The design of SKI ensures correctness, meaning that SKI never causes the kernel to exhibit a behavior that could not possibly occur during normal executions of the kernel, because SKI exercises control over the kernel schedule by temporarily suspending the execution of instructions on chosen CPUs. Correct kernels have to be able to handle this mechanism because the hardware specification does not provide guarantees about the speed of the CPUs. Furthermore, modern kernels are expected to work well within virtual machines, where the apparent speed of CPUs is not guaranteed to be regular simply because the host system might be under heavy load.

Despite this correctness guarantee, some bug detectors may still produce false positives (e.g., data race detectors). In such cases and regardless of how the interleaving space is explored, the obtained results require further analysis specific to the the employed bug detector.

4 Efficiency: Scaling to real code

The total number of possible schedules grows exponentially with the length of the code under test. For most programs, including the kernel, it is not practical to *exhaustively* explore all interleavings, and therefore it is important for concurrency testing tools to include mechanisms for increased scalability.

The p parameter, used by the scheduling algorithm (Section 3.4), constrains the schedules that may be explored and therefore improves scalability by bounding the number of possible schedules. This is done without much impact on the effectiveness of the testing tool, given the observation that, in practice, most bugs can be triggered with few reschedule points [20]. Similarly, it has been shown that many concurrency bugs can be triggered with a small number of threads [48] and with a small number of concurrent requests [60]. Based on these observations, we configured SKI in our tests to use small values for these three dimensions (reschedule points, number of CPUs, and number of system calls).

Despite these optimizations, we noticed in our initial tests that SKI's scalability was limited by the fact that even a single system call can execute a large number of instructions — typical system calls execute many thousands or even millions of instructions. This implied that, even if we limited SKI to $p = 1$, the number of runs that would be required to explore all schedules were on the same order of magnitude as the number of instructions.

To address this scalability issue, SKI relies on a technique first proposed by Godefroid [36] that exploits the fact that some schedules are equivalent and thus redundant, as illustrated in Figure 2. In particular, we rely on the observations that (1) schedules that do not differ in terms of the relative order of communication points (where threads see the effects of each other) are observationally equivalent from the standpoint of the interleaved threads, and that (2) most of the kernel instructions do not constitute communication points between

Schedule 1		Schedule 2		Schedule 3	
CPU1	CPU2	CPU1	CPU2	CPU1	CPU2
A=1		A=1		A=1	
<**inv**>		B=1		B=1	
	A=0	<**inv**>		C=B+A	
	D=A+1		A=0	<**inv**>	
B=1			D=A+1		A=0
C=B+A		C=B+A			D=A+1
PRINT C		PRINT C		PRINT C	

Figure 2: Example showing two equivalent schedules (Schedules 1 and 2) and one schedule that is not equivalent to either of the others (Schedule 3). In this example, only variable `A` is used for communication between CPUs. Because variable `B` is accessed by only one CPU, placing the priority inversion point (`<inv>`) immediately before (Schedule 1) or immediately after (Schedule 2) the statement `B=1` does not change the result of the execution.

CPUs. Taken together, these two observations allow us to significantly improve SKI's scalability by restricting reschedules to occur only at communication points.

More precisely, we define a *point of communication* as an instruction that accesses a memory location that is also accessed by another CPU during the test, and where at least one of the accesses is a write. Such concurrent memory accesses can influence the final outcome of the execution: in the case of two concurrent writes, the last value to be written prevails, and in the case of a write concurrent with a read, the value read may or may not reflect the write, depending on the schedule. Prior tools have also tried to avoid equivalent schedules, but rely instead on identifying and preempting threads at either possible data races or the invocation of synchronization primitives [53].

SKI gathers the location of possible communication points by monitoring memory accesses during the tests. During each run, it tracks the locations of the memory accesses, the CPU responsible for the accesses, and the types of accesses (read or write). After each run, SKI generates a set of program addresses that are potential communication points, and merges this information with an accumulated set of potential communication points for that specific test case. Note that this process does not rely on sample runs — every run monitors the memory accesses and, therefore, potentially learns new communication points. As this accumulated set is constructed, it is used in subsequent runs for the same test case to decide which schedules are equivalent, thereby limiting the set of instructions that qualify as reschedule points.

In our experiments, we observed that, as expected, both data and synchronization accesses were identified as communication points. To give some examples, data accesses occur when both CPUs try to modify the same field in a shared structure (e.g., a file reference count), and synchronization accesses occur when both CPUs try to acquire the same lock. An advantage of SKI's dynamic approach is that whether or not an instruction qualifies as a reschedule point depends on the code that *both* CPUs actually execute (e.g., the specific system calls or interrupt handlers that are invoked). As a result, if two CPUs acquire different locks unrelated to the tested functionality, such accesses will not be considered communication points (in the context of the current test case).

In practice, SKI estimates the expected number of instructions, k (recall Section 3.4), based on previous runs. With the communication points optimization, instead of considering individual instructions when placing reschedule points, we consider only communicating instructions, and thus let the algorithm take coarser-grained steps in its exploration of the interleaving space. That is, by limiting the set of reschedule point candidates, the magnitude of the parameter k is effectively reduced. In addition to these algorithmic optimizations, SKI includes several optimizations, at the level of the implementation, to ensure its effectiveness (Section 5.4).

5 Implementation

We implemented SKI by modifying QEMU, a mature and open-source VMM, and its JIT compiler. In total, our implementation added $13,542$ lines of source code to QEMU. We also built a user-mode testing framework consisting of 674 lines of source code to help users write test cases for SKI (Section 5.3). In addition, we implemented various scripts to set up and automate tests and also to analyze the gathered information.

5.1 Overview

SKI provides a helper tool to allow kernel developers to specify the concurrent system calls, by building a VM containing the corresponding test case (Section 5.3). When executed under SKI, this VM first goes through an initialization phase, performing test-specific actions to configure the system, and then signals the beginning of the test to the VMM using hypercalls (i.e., calls between the VM and the VMM). When all virtual CPUs have received the signal, the SKI scheduler is activated.

SKI's first action is to take a snapshot of the VM. The VM snapshot includes the entire machine state (memory

state, disk state, CPU state, etc.) and thus allows SKI to run multiple executions from an identical initial state.

Starting from this VM snapshot, SKI places reschedule points and assigns starting priorities as described in Sections 3.4 and 4, and then resumes the execution of the highest-priority context and enforces the chosen schedule, thereby exploring different schedules on each run.

To mark the end of the test, the user-mode component inside the VM issues a hypercall to the VMM. Afterwards, the VM is allowed to run normally (i.e., without schedule restrictions) until the testing application asks to terminate the execution. This last phase is useful to let the user-mode component execute test-specific diagnostics (such as a file system check) inside the VM.

5.2 Runnable contexts

The scheduler of SKI allows, at any point in time, only the *live* and *active* context with the highest priority to run. The liveness of a context is inferred by the VMM according to the heuristics explained in Section 3.3; the criteria for determining whether a context is active or not depends on the type of context. A CPU-context is considered active if it has not reached the end of the test, which is flagged by the user-mode component using a hypercall, as discussed above, whereas an interrupt-context is considered active only after it has been triggered by the respective hardware device and before the corresponding IRET instruction has been executed.

5.3 Helper testing framework

We built a user-mode helper framework that allows users to easily build a testing VM ready to be used by SKI. It includes a user-mode application that runs inside the testing VM for the purpose of setting up the kernel and for providing the required test input (e.g., system calls).

The user-mode test framework automatically creates the testing threads/processes, pins each thread/process to a dedicated virtual CPU, issues the hypercalls to mark the beginning of the test (right before the test function is called) and the ending of the test (right after the test function returns), and finally requests the termination of the VM (when all post-test functions have completed). This framework can be used both to manually create test cases (Section 6.3) or to adapt existing test suites to leverage SKI for the interleaving exploration (Section 6.2).

We first implemented the framework targeting Linux and subsequently ported it to FreeBSD, and have been using it to conduct tests on both operating systems. The helper framework itself was easily ported because only few of the system/library calls it relies upon are not part of the POSIX standard (namely the calls to pin threads/processes, which have slightly different interfaces).

5.4 Optimizations and parallelization

In addition to the algorithmic optimizations described in Section 4, we have implemented several other optimizations to improve the performance of SKI. One of our main optimizations avoids resuming from a snapshot for each tested execution, which can take a few seconds in the original version of QEMU. Instead we have implemented in SKI a multi-threaded forking mechanism to take advantage of the copy-on-write semantics offered by the host OS, amortizing the cost of resuming from a snapshot over multiple executions. This benefit is not limited to executions that test the same input because we allow the testing application to receive, through a hypercall, a parameter that specifies the testing input. Thus, from a single snapshot, SKI can explore different inputs and different interleavings, making the overall cost of creating and resuming from a snapshot negligible.

In addition, given that in our testing scenario after each execution we discard most of the state of the VM (e.g. VM RAM and disk contents), we optimized SKI by converting several file system operations, performed by the original QEMU on the host, into memory operations.

Given that our workload is parallelizable, SKI takes advantage of multicore host machines by spawning multiple VMs to perform multiple concurrent tests. We have also implemented a testing infrastructure to distribute the workload across multiple machines, further increasing the testing throughput.

5.5 Bug detectors

Section 3 presented the algorithms and mechanisms that SKI employs to explore the thread interleaving space of the kernel. However, to find concurrency bugs an orthogonal problem needs to be addressed — it is necessary to identify which of these executions triggered bugs.

In Section 6 we show how SKI can be combined with different types of bug detectors — we evaluate SKI using bug detectors to detect crashes, assertion violations, data races and file system inconsistencies. Our implementation detects crashes and assertion violations by monitoring the console output at the VMM level. The detection of data races is also performed at the VMM level by recording racing memory accesses, similarly to DataCollider [32]. File system inconsistencies, in contrast, are detected by running existing file system checkers inside the VM itself after each test.

5.6 Traces and bug diagnosis

To enable the implementation of external bug detectors and to allow the diagnosis of bugs through manual inspection, SKI is able to produce detailed logs of the executions. These traces contain the exact ordering of instructions and the identity of the context responsible for the instructions. In addition, SKI can be configured to produce traces with all the memory accesses and the values of the main CPU registers.

We built some analysis tools that parse these traces to provide useful information. One of our tools produces source code information by disassembling the instructions and by annotating the trace with the source code that generated the instructions (assuming the kernel is compiled with debugging symbols). We also implemented another diagnosis tool that generates the call graph for each execution. While none of these tools is conceptually particularly challenging, in our experience, they complement each other well and make the rich information collected by SKI much more accessible.

Apart from the traces produced by SKI, the bug detectors we built are another important source of diagnostic information. For example, the data race detector that we implemented identifies the exact memory address as well as the instruction addresses involved. As another example, the crash reports produced by the Linux kernel include a detailed stack trace that is very convenient for developers to diagnose bugs.

6 Evaluation

This section evaluates the effectiveness of SKI in revealing real-world kernel concurrency bugs. After describing the configuration that we employed in our experiments, we report our experience in applying SKI to recent and stable versions of the Linux kernel, which resulted in the discovery of several previously unknown concurrency bugs (Section 6.2). We then report on our experiments using SKI to reproduce previously known bugs and comparing it with traditional approaches (Section 6.3).

6.1 Configuration

We conducted our experiments on host machines with dual Intel Xeon X5650 processors and 48 GB of RAM running Linux 3.2.48.1 as the host kernel. To increase the testing throughput, we configured SKI to run 22 testing executions in parallel on each machine and we ran our experiments on up to 12 machines at a time.

For each test case reported in this paper, we configured SKI to use $p = 2$ and we explored 200 schedules in the large-scale experiments to find new bugs (Section 6.2) and $50,000$ schedules in the experiments to reproduce known bugs (Section 6.3). SKI's liveness heuristics used $h_2 = 30$, $h_3 = 20$ and $h_4 = 500,000$ (Section 3.3). We tested several different versions of Linux, ranging from 2.6.28 to 3.13.5, depending on the experiment, and one of the experiments tested FreeBSD, version 8.0. Importantly, the same configuration of SKI was used in all tests: we did not have to modify any settings to adjust SKI to a particular tested kernel version, and we also did not have to modify the kernels under test.

6.2 Finding concurrency bugs

To demonstrate the effectiveness of SKI in finding real world concurrency bugs, we tested several file systems from recent versions of the Linux kernel.

To create the inputs that form the various tests, we modified *fsstress* [44], adding calls to SKI's hypercalls to flag the beginning and the end of the tests, and we modified the test suite to issue concurrent system calls. For convenience we also converted some of the debugging messages to use SKI's own debugging hypercalls. Because one of the file systems (*btrfs*) supports several operations that were not supported by the original *fsstress*, we also added support for twelve of those file system operations (e.g., snapshot/sub-volume operations and dynamic addition/removal of devices). In total, we added or modified 900 lines of code in *fsstress*, of which 700 lines are related to the *btrfs* operations.

6.2.1 Bug detectors

We ran SKI with three bug detectors. The first detector monitors the console output to detect crashes, assertion violations and kernel warning messages. The second detector uses file system checkers (*fsck*), which are specific to each file system and are only supported/mature in the case of some file systems, to detect file system corruption. This bug detector runs inside the VM, in contrast with the others, which are implemented at the VMM level. To limit the performance impact of running *fsck* after each execution, we created small file systems (300 MB) and we mounted the file system in memory using *loop* + *tmpfs* (in addition to the optimizations described in Section 5.4).

The third bug detector consists of a data race detector that we implemented, which analyzes all memory accesses, without sampling. Similarly to other data race detectors [32], our detector finds *racing memory accesses* without distinguishing whether those accesses are performed by synchronization functions. The main chal-

Bug	Kernel	FS	Function	Detector / Failure	E	FS	Status
1	3.11.1	Btrfs	btrfs_find_all_root()	Crash: Null-pointer	41	0.030	Fixed
2	3.11.1	Btrfs	run_clustered_refs()	Crash: Null-pointer + Warning	26	0.020	Fixed
3	3.11.1	Btrfs	record_one_backref()	Warning	74	0.030	Fixed
4	3.11.1	Btrfs	NA	Fsck: Refs. not found	11	0.200	Reported
5	3.12.2+p	Btrfs	btrfs_find_all_root()	Crash: Null pointer	61	0.060	Fixed
6	3.12.2	Btrfs	inode_tree_add()	Warning	53	0.010	Fixed
7	3.13.5	Logfs	indirect_write_alias()	Crash: Null pointer	31	0.065	Reported
8	3.13.5	Logfs	btree_write_alias()	Crash: Invalid paging	142	0.020	Reported
9	3.13.5	Jfs	lbmIODone()	Crash: Assertion	74	0.005	Reported
10	3.13.5	Ext4	ext4_do_update_inode()	Data race	32	0.005	Fixed
11	3.13.5	VFS	generic_fillattr()	Data race	125	0.005	Reported

Table 1: Bugs that have been discovered by SKI in recent versions of the Linux kernel and that we have reported to developers. For the specific input that triggered each bug, we show the number of schedules that were required to expose the bug (E) and the fraction of schedules that triggered the bug (FS). Eventually we found out that bug #3 had previously been reported. A patched version of the kernel, expected to solve bug #1, was tested on request from the developers but SKI revealed that the kernel could still crash in a different location of the same function (bug #5).

		Reports
False data race		76
Data race	Benign	53
	Under investigation	37
	Harmful	24

Table 2: Types of race reports found during our experiments. The numbers displayed refer to the number of reports after associating related races. Note that a single bug may be involved in multiple data races (e.g., if it affects multiple variables).

	Btrfs	Ext4	Jfs	Logfs
SKI	34.7	62.6	61.6	61.2
SKI+ DR	32.1	61.9	59.5	58.8
SKI+ Fsck	6.4	20.8	18.2	N/A
SKI+ Fsck + DR	6.1	20.6	17.9	N/A

Table 3: SKI's throughput (for each machine) with different bug detectors. Throughput is given in thousands of executions per hour. DR denotes the data race detector. Fsck tests on *logfs* are absent due to the lack of compatible mature checkers.

lenge in this case is filtering out the false positives (*false* data races and *benign* data races) [32, 51, 62, 78]. In order to facilitate this manual process, our tool groups together distinct pairs or racing instructions that were found to race directly or transitively. Using this method, we we were able to group together 3114 pairs of races into 190 race reports. Filtering out race reports that were not data races was straightforward, but the difficult part was separating real data races into benign and harmful ones. In some cases, this requires careful analysis of the code and documentation and, ultimately, it may require asking the developers – who may not even agree among themselves. Heuristics could have been used to analyze the results, but unfortunately these typically offer limited help for the more complicated cases. Given this complexity, we gathered some reports (not included in Table 1) that may constitute bugs but are still under analysis, and for which, in some cases, we are still waiting for feedback from the developers. Table 2 shows the number of race reports that we obtained in the file systems tests according to their type.

6.2.2 Results

The results in Figure 1 show that SKI was able to find several unknown concurrency bugs in mature versions of the Linux kernel. One of the bugs found affects the widely used *ext4* file system and six bugs affect the *btrfs* file system – which is expected to soon become the default file system in some distributions [8]. We have reported the 11 bugs listed in Table 1; of those, 6 have already been fixed.

Furthermore, although FS related system calls tend to be expensive, SKI was able to achieve a testing throughput that reached 62 thousand executions per hour on each machine (Table 3). Even though the current performance of SKI proved to be effective, significant performance improvements may still be achievable by using more efficient virtual machines monitors, possibly using hardware acceleration, or even by building SKI using binary

Bug	Kernel	OS Component	Failure	E	FS
A [4]	Linux 2.6.28	Anonymous pipes	Crash	28	0.00572
B [5]	Linux 3.2	Inotify + FAT32	Crash	53	0.13770
C [9]	Linux 3.6.1	Proc file system + Ext4	Semantic	51	0.01004
D [3]	FreeBSD 8.0	Sockets	Semantic	3519	0.00014

Table 4: Known bugs reproduced with SKI. The table shows the number of schedules that were required to expose the bug (E) and the fraction of schedules that triggered the bug (FS). The table shows the kernel version under which we reproduced the bug, the OS components involved and the type of failure that the bug causes.

instrumentation frameworks.

It is worth pointing out that many of the bugs found by SKI are serious – six of the bugs cause the kernel to crash and most of the bugs found cause persistent data loss. For example, the *ext4* bug, which is due to improper synchronization while updating the inodes, causes the field *i_disksize* (containing information about the size of the inodes) to become corrupted. To fix this bug, developers applied patches that involved refactoring the code and the introduction of additional synchronization.

6.3 Reproducing concurrency bugs

We also evaluated the effectiveness of SKI in reproducing previously reported kernel concurrency bugs. To find typical bug reports, we searched the kernel Bugzilla databases, the kernel development histories (i.e., the *git changelogs*), and the mailing list archives. From these sources, we selected four independently confirmed kernel concurrency bugs. We opted for a diverse set of bugs that were particularly well documented. Furthermore, to enable a direct comparison, we considered only bug reports that included instructions for triggering the reported bugs through stress testing.

As listed in Table 4, the selected bugs exhibited different types of failures in various kernel components. Bug A causes a memory access violation (an "Oops" in Linux parlance) in the *pipe* communication mechanism, which can occur during concurrent *open* and *close* calls on anonymous pipes. Bug B also results in a memory access violation and is triggered on some interleavings when a FAT32-formatted partition is unmounted concurrently with the removal of an *inotify* watch[4] associated with the same partition. Bug C does not result in a crash, but rather causes a *read* system call to return corrupted values. Finally, bug D affects FreeBSD and is triggered by concurrent calls on sockets that cause the kernel to incorrectly return error values.

[4]Linux's *inotify* interface allows processes to receive change notifications for file system objects such as files, directories, or mount points.

Based on these four bug reports, we determined the system calls that would expose the bugs and produced the corresponding SKI test cases, as described in Section 5.3. For the bugs that had semantic manifestations, i.e., system calls that returned wrong results, we implemented bug-specific detectors, according to the information provided in the bug reports.

SKI exposed bugs A and B by triggering the crash after exploring 28 and 53 schedules, respectively. Bugs C and D were exposed after 51 and 3519 schedules, respectively, causing wrong results to be returned. Given that SKI requires few executions to trigger concurrency bugs, with a suitable test suite (e.g. regression test suites [38]), SKI's throughput is sufficient to reproduce on the order of hundreds of such concurrency bugs per hour (Table 5).

These experiments confirm that SKI is effective at reproducing real-world concurrency bugs. Most importantly, it should be noted that the reproduced bugs stem from two different OS code bases (FreeBSD and Linux) and from a wide range of Linux kernel versions spanning several years of intense development. In fact, even if we ignore the cumulative number of lines changed (i.e., the churn rate) and take into consideration only the increase in the total number of lines of source code, the Linux kernel grew by an impressive 60% from version 2.6.28 (10M SLOC) to version 3.6.1 (16M SLOC). SKI handled the different versions of the Linux kernel and the FreeBSD kernel without requiring any changes to the VMM itself or its configuration, which provides evidence for the considerable versatility intrinsic to SKI's design.

6.3.1 Comparison with stress testing

In the discussions that led to the resolution of these four bugs, the kernel developers proposed non-systematic methods to reproduce them. In particular, they provided simple stress tests, which continuously execute the same operations in a tight loop, waiting until a buggy interleaving occurs. We executed the original stress tests proposed by the developers to compare SKI to a traditional approach. For this purpose, we ran the stress tests in an

Bug	Throughput
A	302.0
B	169.3
C	218.7
D	501.4

Table 5: SKI's throughput for each machine. Throughput is presented in thousand executions per hour.

unmodified VMM, i.e., without making use of SKI.

Note that without a deep knowledge of the kernel code, in the general case, it is hard to generate stress tests for the bugs that SKI discovered in Section 6.2. The reason for this is that it is not straightforward to ensure that, for every one of the various iterations of the stress test, the state of the kernel is such that it can trigger the concurrency bugs. (SKI avoids this problem because it automatically restores the initial state through snapshotting). Thus, to ensure a more objective comparison between the two approaches, we chose to use stress tests produced by the kernel developers themselves since these are the ones offering better effectiveness guarantees.

As expected, and consistent with earlier comparisons of systematic and unsystematic user-mode concurrency testing approaches [20, 53], SKI proved to be much more effective in reproducing concurrency bugs than the non-systematic approaches. Despite the fact that we gave each stress test up to 24 hours to complete, bug A and bug D were not triggered at all by their corresponding stress tests. While the stress tests for bugs B and C did eventually trigger their corresponding bugs, they required significantly more executions (and time) than SKI: the stress tests required more than 200,000 iterations (4 hours) to reproduce bug B and more than 800 iterations (1 minute) to trigger bug C, compared to 53 and 51 iterations (both a few seconds), respectively, under SKI.

Overall, the relative difficulty of reproducing bugs with simple stress tests is not surprising given prior comparisons of systematic approaches and stress testing in the context of user-mode applications [20]. Furthermore, this difficulty was also reported by the kernel developers themselves. For example, in the case of bug A (which the stress test failed to reproduce in our experiments) the developer stated that the "failure window is quite small" [6] and recommended introducing a carefully placed sleep statement in the kernel to trigger the bug.

6.3.2 Liveness heuristics

We instrumented SKI to log the activation of SKI's heuristics. Using this data we calculated the percentage of schedules that triggered each of the heuristics (Table 6) and the average number of times each heuristic was triggered per schedule (Table 7).

Bug	H1	H2	H3	H4	H*
A	1.72%	0.61%	5.71%	0.57%	7.97%
B	88.80%	49.70%	0.05%	13.73%	88.93%
C	1.50%	23.56%	0.00%	0.00%	25.06%
D	0.53%	2.66%	0.00%	0.00%	3.05%

Table 6: Percentage of schedules that triggered the liveness heuristics. H* refers to the percentage of schedules that trigger any heuristic.

Bug	H1	H2	H3	H4	H*
A	0.08	0.01	0.06	0.01	0.17
B	14.97	1.59	36.38	0.14	53.08
C	0.01	0.44	0.00	0.00	0.45
D	0.01	0.03	0.00	0.00	0.03

Table 7: Average number of times the liveness heuristics were triggered per schedule. H* refers to the percentage of schedules that trigger any heuristic.

The results show that some of the schedules do not trigger heuristics. This is expected to happen when SKI chooses schedules in which threads do not experience lock contention and is more likely to occur in operating systems that are well optimized for scalability. Even though not all of the tests activate all heuristics, all heuristics were activated in at least one of the test cases.

In addition, we observed that in these tests the heuristics were triggered at 167 distinct instruction addresses. The large number of distinct addresses is indicative of the challenges that would result from manually annotating the kernel to infer thread liveness, as opposed to relying on the four simple heuristics implemented by SKI.

6.3.3 Effectiveness of communication points

To evaluate the effectiveness of the optimization of keeping track of communication points and allowing reschedules to occur only at these points (described in Section 4), we calculated for each test case the average number of instructions and the average number of communication points executed per run. As shown in Table 8, this optimization reduced the number of potential reschedule points by up to an order of magnitude in our experiments, thereby avoiding the wasteful exploration of redundant, effectively equivalent schedules. This shows the importance of this optimization to the scalability of SKI.

Bug	I	CP	I/CP
A	87673.5	12511.2	7.00
B	210693.0	23432.8	8.99
C	65126.9	6372.3	10.22
D	22641.3	6503.2	3.48

Table 8: Effectiveness of the communication points optimization described in Section 4. The table shows for each reproduced bug the average number of instructions executed per run (I) and the average communication points executed per run (CP). The last column characterizes the optimization's effectiveness as the ratio of the two metrics.

7 Discussion

SKI proposes a VMM-based scheduler. In this section, we discuss some of the implications of this choice.

A limitation of relying on a VMM is that the kernel running inside a virtual machine is limited to using the hardware virtualized by the VMM. For a testing tool, it means that it is not possible to reproduce bugs that require hardware that is not virtualized by the VMM. However, we believe this does not detract significantly from SKI's practical value because the size of the device-independent kernel core is already considerable. Further, it may be possible to overcome the VMM dependency by building an equivalent tool based on kernel binary instrumentation, which is an active area of research [33].

The choice of a VMM-based approach has another important consequence. Because the VMM emulates one instruction at a time, and propagates its effects to all other CPUs immediately afterwards, concurrency bugs that arise from wrongly assuming a strong memory model are not necessarily exposed. This is because some CPUs offer weaker memory models, which can have very complex semantics, to the point where official specifications have been found to not match the observed semantics of hardware [11]. This is a complex problem — significant effort has been directed at simply studying the semantics of CPUs with relaxed memory models [61] — and we believe that effectively diagnosing this type of concurrency bug will likely require more specialized tools. Such bugs are currently not the target of SKI.

8 Related Work

Schedule space exploration. The traditional way to test applications for concurrency bugs relies on manually created stress tests. To increase the chances of unmasking concurrency bugs, researchers have proposed various tools that rely on introducing sleeps in the code to disrupt the scheduling of threads [14, 19, 32, 56, 63, 64]. The common limitation to these approaches is that they do not systematically explore the thread interleaving space.

To address these limitations, a different class of tools has been proposed to test for concurrency bugs [20, 53, 54]. This approach relies on taking full control of the scheduling of threads to avoid redundant interleavings and therefore increases the effectiveness of testing [20]. A previous attempt [17] to systematically test kernel code has focused on small-scale educational kernels and relied on modifications to the tested kernels. SKI follows the systematic approach, but distinguishes itself from existing tools by being applicable to kernel code and by being scalable to real-world kernels.

Because the schedule space is extremely large, systematic tools take advantage of different techniques to restrict the schedule exploration while still ensuring effectiveness. Examples used in the context of finding user-mode concurrency bugs include preemption bounding [53], reschedule bounding [20] and the elimination of redundant schedules [36]. Other work has proposed limiting the valid run-time schedules by reducing or eliminating the schedule non-determinism [27–30, 46, 71, 75]. Restricting the kernel schedules by applying these techniques could further increase the effectiveness of SKI.

Symbolic execution [21, 25] is an analysis technique that systematically explores the application execution path space by keeping track, during execution, of symbolic values instead of concrete values. Symbolic execution has been applied to multi-threaded applications by implementing a custom user-mode scheduler [41]. More recently, SymDrive [57] has been successful at testing kernel device drivers using symbolic execution, although it requires modifications to the kernel and does not target concurrency bugs. Similarly, SWIFT [22] uses symbolic execution to test kernel file system checkers but does not target concurrency bugs. By using SKI's ability to instrument kernel schedules, it may be possible to leverage the symbolic execution approach in the context of testing the kernel for concurrency bugs.

Similarly to shared-memory systems, which are the focus of SKI, distributed systems are also prone to schedule-dependent bugs [45, 47, 58] and the complexity of distributed systems also requires dedicated techniques to scale to real-world applications. For example, CrystalBall [73] proposes model checking live systems and steering their execution away from states that trigger bugs. By exploring states based on snapshots of live systems, CrystalBall is able to explore states that are more likely to be relevant to the current execution than con-

ducting the entire exploration from a single initial state. MoDist [74] also finds bugs in distributed systems but does so transparently, without requiring implementations to be written in special languages. MoDist is able to scale to complex implementations by judiciously simulating events that typically trigger bugs, such as the reordering of messages and the expiration of timers.

Detecting concurrency bugs. Different types of bug detectors have been proposed to detect at runtime whether an execution is anomalous [2, 18, 32, 34, 35, 49, 51, 62, 68, 78]. Detecting anomalous executions is a challenge complementary to the exploration of the interleaving space. DataCollider [32] is a kernel data race detector that randomly pauses CPUs, through the use of hardware breakpoints, to cause non-systematic schedule diversity. In Section 5.5 we show how we combined DataCollider's data race detection mechanism with SKI to detect data races. RedFlag [66] is another example of a concurrency bug detector for the kernel that combines a block-based atomicity checker with a lockset based data race detector. In Section 5.5, we describe in detail how SKI leverages various bug detectors.

Deterministic replay. Determinism is valuable for diagnosing concurrency bugs [13, 15, 27, 28, 30, 46], but ensuring determinism is orthogonal to the systematic interleaving exploration problem. Given the same, fixed input parameters, SKI, like its user-mode counterparts, can deterministically re-execute the same schedule, provided the kernel is given identical input in each run. Currently, SKI does not ensure that the same hardware input is provided to the kernel (e.g. low-granularity timer values). Input determinism could be achieved through the use of another layer running below the VMM [46] or by modifying the VMM [31, 72].

Input space exploration. Dynamic testing techniques require running the tested software and providing it with testing input. The traditional approach has relied on manually writing test cases [38], but more sophisticated approaches have been proposed to address this challenge. Such approaches include blackbox fuzzers [16], semantically-aware fuzzers [1, 10] and symbolic execution techniques [21, 37, 42]. Because file systems have a particularly large input space and are critical components in the system, file system testing has been a particularly active area of research [12, 22, 50, 76, 77]. Even though the focus of SKI is on the exploration of the interleaving space, to evaluate SKI we explored the kernel input space with an existing file system test suite, *fsstress* [44].

Virtual machine introspection (VMI). Several VMM mechanisms have been proposed to infer high-level information of virtual machines [23]. In many cases the purpose of these mechanisms is to increase performance. Examples include improving the host memory usage by inferring which guest memory is actively being used [24], improving IO performance by anticipating IO requests [40] and improving the scalability of virtual machine monitors by inferring whether the virtual machine is executing critical sections [65, 70]. In addition, VMI techniques have been leveraged to gather information about virtual machines in security contexts [55]. Using the introspection approach, SKI infers the liveness of threads for the purpose of achieving fine-level control over the threads schedules. For example, SKI leverages the observation that the PAUSE instruction is typically associated with spin-locks, as does the work of Wang et. al [70] in the context of increasing VMM performance.

9 Conclusion

This paper introduces SKI, the first practical testing tool to systematically explore the interleaving space of real-world kernel code. SKI does not require any modifications to tested kernels, nor does it require knowledge of the semantics of any kernel synchronization primitives. We detailed key optimizations that make SKI scale to real-world code, and we have shown that SKI is effective at finding buggy schedules in both FreeBSD and various versions of the Linux kernel, without changing or annotating the tested kernel.

As future work, we plan to explore different bug detectors and to leverage the control provided by SKI to effectively explore the input space.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback and our shepherd, Junfeng Yang, for his help. Pedro Fonseca was partially supported by a fellowship from the Portuguese Foundation for Science and Technology (FCT). Rodrigo Rodrigues was funded by the European Research Council under an ERC starting grant.

References

[1] A Linux System call fuzz tester. http://codemonkey.org.uk/projects/trinity/.

[2] ANNOUNCE: Lock validator. http://lwn.net/Articles/185605/.

[3] Bug 144061 - [socket] race on unix socket close. https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=144061.

[4] Bug 14416 - Null pointer dereference in fs/pipe.c . http://bugzilla.kernel.org/show_bug.cgi?id=14416.

[5] Bug 22602 - Oops while unmounting an USB key with a FAT filesystem. https://bugzilla.kernel.org/show_bug.cgi?id=22602.

[6] FS: pipe.c null pointer dereference. https://git.kernel.org/cgit/linux/kernel/git/stable/stable-queue.git/tree/queue-2.6.31/fs-pipe.c-null-pointer-dereference.patch?id=36e97dec52821f76536a25b763e320eb7434c2a5.

[7] Kernel threads made easy. http://lwn.net/Articles/65178/.

[8] OpenSUSE News. https://news.opensuse.org/2014/03/19/development-for-13-2-kicks-off/.

[9] Patch "ext4: fix crash when accessing /proc/mounts concurrently" has been added to the 3.6-stable tree. http://www.mail-archive.com/stable@vger.kernel.org/msg19380.html.

[10] AITEL, D. The advantages of block-based protocol analysis for security testing. Tech. rep., Immunity, Inc., 2002.

[11] ALGLAVE, J., FOX, A., ISHTIAQ, S., MYREEN, M. O., SARKAR, S., SEWELL, P., AND NARDELLI, F. Z. The semantics of POWER and ARM multiprocessor machine code. In *Proc. of Workshop on Declarative Aspects of Multicore Programming (DAMP)* (2008).

[12] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., BAIRAVASUNDARAM, L. N., DENEHY, T. E., POPOVICI, F. I., PRABHAKARAN, V., AND SIVATHANU, M. Semantically-smart disk systems: Past, present, and future. *SIGMETRICS Perform. Eval. Rev. 33*, 4 (Mar. 2006), 29–35.

[13] AVIRAM, A., WENG, S.-C., HU, S., AND FORD, B. Efficient system-enforced deterministic parallelism. In *Proc. of Operating System Design and Implementation (OSDI)* (2010).

[14] BEN-ASHER, Y., EYTANI, Y., FARCHI, E., AND UR, S. Noise makers need to know where to be silent – Producing schedules that find bugs. In *Proc. of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)* (2006).

[15] BERGER, E. D., YANG, T., LIU, T., AND NOVARK, G. Grace: Safe multithreaded programming for C/C++. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2009).

[16] BIRD, D. L., AND MUNOZ, C. U. Automatic generation of random self-checking test cases. *IBM Syst. J. 22*, 3 (Sept. 1983), 229–245.

[17] BLUM, B. *Landslide: Systematic Dynamic Race Detection in Kernel Space*. MS thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 2012. http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-CS-12-118.pdf.

[18] BOND, M. D., COONS, K. E., AND MCKINLEY, K. S. PACER: Proportional detection of data races. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2010).

[19] BRON, A., FARCHI, E., MAGID, Y., NIR, Y., AND UR, S. Applications of synchronization coverage. In *Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2005).

[20] BURCKHARDT, S., KOTHARI, P., MUSUVATHI, M., AND NAGARAKATTE, S. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2010).

[21] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of Operating System Design and Implementation (OSDI)* (2008).

[22] CARREIRA, J. A., RODRIGUES, R., CANDEA, G., AND MAJUMDAR, R. Scalable testing of file system checkers. In *Proc. of European Conference on Computer Systems (EuroSys)* (2012).

[23] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *Proc. of Hot Topics in Operating Systems (HotOS)* (2001).

[24] CHIANG, J.-H., LI, H.-L., AND CHIUEH, T.-C. Introspection-based memory de-duplication and migration. In *Proc. of International Conference on Virtual Execution Environments (VEE)* (2013).

[25] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[26] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. RadixVM: Scalable address spaces for multithreaded applications. In *Proc. of European Conference on Computer Systems (EuroSys)* (2013).

[27] CUI, H., SIMSA, J., LIN, Y.-H., LI, H., BLUM, B., XU, X., YANG, J., GIBSON, G. A., AND BRYANT, R. E. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proc. of Symposium on Operating System Principles (SOSP)* (2013).

[28] CUI, H., WU, J., GALLAGHER, J., GUO, H., AND YANG, J. Efficient deterministic multithreading through schedule relaxation. In *Proc. of Symposium on Operating System Principles (SOSP)* (2011).

[29] CUI, H., WU, J., TSAI, C.-C., AND YANG, J. Stable deterministic multithreading through schedule memoization. In *Proc. of Operating System Design and Implementation (OSDI)* (2010).

[30] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. DMP: Deterministic shared memory multiprocessing. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009).

[31] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *Proc. of International Conference on Virtual Execution Environments (VEE)* (2008).

[32] ERICKSON, J., MUSUVATHI, M., BURCKHARDT, S., AND OLYNYK, K. Effective data-race detection for the kernel. In *Proc. of Operating System Design and Implementation (OSDI)* (2010).

[33] FEINER, P., BROWN, A. D., AND GOEL, A. Comprehensive kernel instrumentation via dynamic binary translation. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[34] FLANAGAN, C., AND FREUND, S. N. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. of Symposium on Principles of Programming Languages (POPL)* (2004).

[35] FONSECA, P., LI, C., AND RODRIGUES, R. Finding complex concurrency bugs in large multi-threaded applications. In *Proc. of European Conference on Computer Systems (EuroSys)* (2011).

[36] GODEFROID, P. Model checking for programming languages using VeriSoft. In *Proc. of Symposium on Principles of Programming Languages (POPL)* (1997).

[37] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. *SIGPLAN Not. 40*, 6 (2005), 213–223.

[38] GRAVES, T. L., HARROLD, M. J., KIM, J.-M., PORTER, A., AND ROTHERMEL, G. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol. 10*, 2 (Apr. 2001), 184–208.

[39] HOLZMANN, G. J. Mars code. *Commun. ACM 57*, 2 (2014), 64–73.

[40] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Antfarm: Tracking processes in a virtual machine environment. In *Proc. of Annual Technical Conference (ATC)* (2006).

[41] KASIKCI, B., ZAMFIR, C., AND CANDEA, G. Data races vs. data race bugs: Telling the difference with portend. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[42] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[43] KOEHNEMANN, H., AND LINDQUIST, T. E. Towards target-level testing and debugging tools for embedded software. In *TRI-Ada* (1993).

[44] LARSON, P. Testing linux with linux test project. In *Proc. of Ottawa Linux Symposium (OLS)* (2002).

[45] LI, S., ZHOU, H., LIN, H., XIAO, T., LIN, H., LIN, W., AND XIE, T. A characteristic study on failures of production distributed data-parallel programs. In *Proc. of International Conference on Software Engineering (ICSE)* (2013).

[46] LIU, T., CURTSINGER, C., AND BERGER, E. D. Dthreads: Efficient deterministic multithreading. In *Proc. of Symposium on Operating System Principles (SOSP)* (2011).

[47] LIU, X. WiDS checker: Combating bugs in distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)* (2007).

[48] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008).

[49] LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. AVIO: detecting atomicity violations via access interleaving invariants. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006).

[50] MA, A., DRAGGA, C., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Ffsck: The fast file system checker.

[51] MARINO, D., MUSUVATHI, M., AND NARAYANASAMY, S. LiteRace: Effective sampling for lightweight data-race detection. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2009).

[52] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Proc. of International Conference on Parallel and Distributed Computing and Systems (PDCS)* (1998).

[53] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *Proc. of Operating System Design and Implementation (OSDI)* (2008).

[54] NAGARAKATTE, S., BURCKHARDT, S., MARTIN, M. M., AND MUSUVATHI, M. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2012).

[55] NANCE, K., BISHOP, M., AND HAY, B. Virtual machine introspection: Observation or interference? *IEEE Security and Privacy 6*, 5 (Sept. 2008), 32–37.

[56] PARK, S., LU, S., AND ZHOU, Y. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009).

[57] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. SymDrive: Testing drivers without devices. In *Proc. of Operating System Design and Implementation (OSDI)* (2012).

[58] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)* (2006).

[59] RUSSELL, P. R. Unreliable Guide To Locking. `http://kernelbook.sourceforge.net/kernel-locking.pdf`.

[60] SAHOO, S. K., CRISWELL, J., AND ADVE, V. S. An empirical study of reported bugs in server software with implications for automated bug diagnosis. Tech. Report 2142/13697, University of Illinois, 2009.

[61] SARKAR, S., SEWELL, P., NARDELLI, F. Z., OWENS, S., RIDGE, T., BRAIBANT, T., MYREEN, M. O., AND ALGLAVE, J. The semantics of x86-CC multiprocessor machine code. In *Proc. of Symposium on Principles of Programming Languages (POPL)* (2009).

[62] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multi-threaded programs. In *Proc. of Symposium on Operating System Principles (SOSP)* (1997).

[63] SEN, K. Race directed random testing of concurrent programs. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2008).

[64] STOLLER, S. D. Testing concurrent Java programs using randomized scheduling. In *Proc. of Workshop on Runtime Verification (RV)* (2002).

[65] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *Proc. of Conference on Virtual Machine Research And Technology Symposium (VM)* (2004).

[66] URTEAGA, I. N., BARNHART, K., AND HAN, Q. REDFLAG: A run-time, distributed, flexible, lightweight, and generic fault detection service for data-driven wireless sensor applications. *Pervasive Mob. Comput. 5* (October 2009), 432–446.

[67] VALOIS, J. D. Implementing lock-free queues. In *Proc. of International Conference on Parallel and Distributed Computing and Systems (PDCS)* (1994).

[68] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proc. of Symposium on Operating System Principles (SOSP)* (2011).

[69] WAGNER, S., JÜRJENS, J., KOLLER, C., AND TRISCHBERGER, P. Comparing bug finding tools with reviews and tests. In *Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (Testcom)* (2005).

[70] WANG, Z., LIU, R., CHEN, Y., WU, X., CHEN, H., ZHANG, W., AND ZANG, B. COREMU: A scalable and portable parallel full-system emulator. In *Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2011).

[71] WU, J., TANG, Y., HU, G., CUI, H., AND YANG, J. Sound and precise analysis of parallel programs through schedule specialization. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2012).

[72] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. of Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)* (2007).

[73] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)* (2009).

[74] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)* (2009).

[75] YANG, J., CUI, H., WU, J., TANG, Y., AND HU, G. Determinism is not enough: Making parallel programs reliable with stable multithreading. *Communications of the ACM* (2014).

[76] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proc. of Operating System Design and Implementation (OSDI)* (2006).

[77] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *Proc. of Operating System Design and Implementation (OSDI)* (2004).

[78] YU, Y., RODEHEFFER, T., AND CHEN, W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proc. of Symposium on Operating System Principles (SOSP)* (2005).

All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications

Thanumalayan Sankaranarayana Pillai Vijay Chidambaram Ramnatthan Alagappan
Samer Al-Kiswany Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau

University of Wisconsin-Madison

Abstract

We present the first comprehensive study of application-level crash-consistency protocols built atop modern file systems. We find that applications use complex update protocols to persist state, and that the correctness of these protocols is highly dependent on subtle behaviors of the underlying file system, which we term *persistence properties*. We develop a tool named BOB that empirically tests persistence properties, and use it to demonstrate that these properties vary widely among six popular Linux file systems. We build a framework named ALICE that analyzes application update protocols and finds *crash vulnerabilities*, i.e., update protocol code that requires specific persistence properties to hold for correctness. Using ALICE, we analyze eleven widely-used systems (including databases, key-value stores, version control systems, distributed systems, and virtualization software) and find a total of 60 vulnerabilities, many of which lead to severe consequences. We also show that ALICE can be used to evaluate the effect of new file-system designs on application-level consistency.

1 Introduction

Crash recovery is a fundamental problem in systems research [8, 21, 34, 38], particularly in database management systems, key-value stores, and file systems. Crash recovery is hard to get right; as evidence, consider the ten-year gap between the release of commercial database products (e.g., System R [7, 8] and DB2 [34]) and the development of a working crash recovery algorithm (ARIES [33]). Even after ARIES was invented, another five years passed before the algorithm was proven correct [24, 29].

The file-systems community has developed a standard set of techniques to provide file-system metadata consistency in the face of crashes [4]: logging [5, 9, 21, 37, 45, 51], copy-on-write [22, 30, 38, 44], soft updates [18], and other similar approaches [10, 16]. While bugs remain in the file systems that implement these methods [28], the core techniques are heavily tested and well understood.

Many important applications, including databases such as SQLite [43] and key-value stores such as LevelDB [20], are currently implemented on top of these file systems instead of directly on raw disks. Such data-management applications must also be crash consistent, but achieving this goal atop modern file systems is challenging for two fundamental reasons.

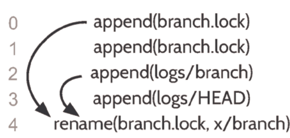
The first challenge is that the exact guarantees provided by file systems are unclear and underspecified. Applications communicate with file systems through the POSIX system-call interface [48], and ideally, a well-written application using this interface would be crash-consistent on any file system that implements POSIX. Unfortunately, while the POSIX standard specifies the effect of a system call in memory, specifications of how disk state is mutated in the event of a crash are widely misunderstood and debated [1]. As a result, each file system persists application data slightly differently, leaving developers guessing.

To add to this complexity, most file systems provide a multitude of configuration options that subtly affect their behavior; for example, Linux ext3 provides numerous journaling modes, each with different performance and robustness properties [51]. While these configurations are useful, they complicate reasoning about exact file system behavior in the presence of crashes.

The second challenge is that building a high-performance application-level crash-consistency protocol is not straightforward. Maintaining application consistency would be relatively simple (though not trivial) if all state were mutated synchronously. However, such an approach is prohibitively slow, and thus most applications implement complex *update protocols* to remain crash-consistent while still achieving high performance. Similar to early file system and database schemes, it is difficult to ensure that applications recover correctly after a crash [41, 47]. The protocols must handle a wide range of corner cases, which are executed rarely, relatively untested, and (perhaps unsurprisingly) error-prone.

In this paper, we address these two challenges directly, by answering two important questions. The first question is: what are the behaviors exhibited by modern file systems that are relevant to building crash-consistent applications? We label these behaviors *persistence properties* (§2). They break down into two global categories: the *atomicity* of operations (e.g., does the file system ensure that `rename()` is atomic [32]?), and the *ordering* of operations (e.g., does the file system ensure that file creations are persisted in the same order they were issued?).

To analyze file system persistence properties, we de-



Vulnerable on file systems:
ext2, ext3-writeback, ext4-writeback, ext4-ordered, btrfs, xfs, reiserfs-nolog, reiserfs-writeback
Not vulnerable on file systems:
ext3-ordered, ext3-journal, ext4-nodelalloc, ext4-journal, xfs-wsync, reiserfs-ordered,reiserfs-journaled

Figure 1: **Git Crash Vulnerability.** *The figure shows part of the Git update protocol. The arrows represent ordering dependencies: if the appends are not persisted before the rename, any further commits to the repository fail. We find that, whether the protocol is vulnerable or not varies even between configurations of the same file system.*

velop a simple tool, known as the *Block Order Breaker* (BOB). BOB collects block-level traces underneath a file system and re-orders them to explore possible on-disk crash states that may arise. With this simple approach, BOB can find which persistence properties do *not* hold for a given system. We use BOB to study six Linux file systems (ext2, ext3, ext4, reiserfs, btrfs, and xfs) in various configurations. We find that persistence properties vary widely among the tested file systems. For example, appends to file A are persisted before a later rename of file B in the ordered journaling mode of ext3, but not in the same mode of ext4, unless a special option is enabled.

The second question is: do modern applications implement crash consistency protocols correctly? Answering this question requires understanding update protocols, no easy task since update protocols are complex [47] and spread across multiple files in the source code. To analyze applications, we develop ALICE, a novel framework that enables us to systematically study application-level crash consistency (§3). ALICE takes advantage of the fact that, no matter how complex the application source code, the update protocol boils down to a sequence of file-system related system calls. By analyzing permutations of the system-call trace of workloads, ALICE produces *protocol diagrams*: rich annotated graphs of update protocols that abstract away low-level details to clearly present the underlying logic. ALICE determines the exact persistence properties assumed by applications as well as flaws in their design.

Figure 1 shows an example of ALICE in action. The figure shows a part of the update protocol of Git [26]. ALICE detected that the appends need to be persisted before the rename; if not, any future commits to the repository fail. This behavior varies widely among file systems: a number of file-system features such as delayed allocation and journaling mode determine whether file systems exhibit this behavior. Some common configurations like ext3 ordered mode persist these operations in order, providing a false sense of security to the developer.

We use ALICE to study and analyze the update protocols of eleven important applications: LevelDB [20], GDBM [19], LMDB [46], SQLite [43], PostgreSQL [49], HSQLDB [23], Git [26], Mercurial [31]), HDFS [40], ZooKeeper [3], and VMWare Player [52]. These applications represent software from different domains and at varying levels of maturity. The study focuses on file-system behavior that affects users, rather than on strictly verifying application correctness. We hence consider typical usage scenarios, sometimes checking for additional consistency guarantees beyond those promised in the application documentation. Our study takes a pessimistic view of file-system behavior; for example, we even consider the case where renames are not atomic on a system crash.

Overall, we find that application-level consistency in these applications is highly sensitive to the specific persistence properties of the underlying file system. In general, if application correctness depends on a specific file-system persistence property, we say the application contains a *crash vulnerability*; running the application on a different file system could result in incorrect behavior. We find a total of 60 vulnerabilities across the applications we studied; several vulnerabilities have severe consequences such as data loss or application unavailability. Using ALICE, we also show that many of these vulnerabilities (roughly half) manifest on current file systems such as Linux ext3, ext4, and btrfs.

We find that many applications implicitly expect *ordering* among system calls (e.g., that writes, even to different files, are persisted in order); when such ordering is not maintained, 7 of the 11 tested applications have trouble properly recovering from a crash. We also find that 10 of the 11 applications expect *atomicity* of file-system updates. In some cases, such a requirement is reasonable (e.g., a single 512-byte write or file rename operation are guaranteed to be atomic by many current file systems when running on a hard-disk drive); in other situations (e.g., with file appends), it is less so. We also note that some of these atomicity assumptions are not future proof; for example, new storage technology may be atomic only at a smaller granularity than 512-bytes (e.g., eight-byte PCM [12]). Finally, for 7 of the 11 applications, *durability* guarantees users likely expect are not met, often due to directory operations not being flushed.

ALICE also enables us to determine whether new file-system designs will help or harm application protocols. We use ALICE to show the benefits of an ext3 variant we propose (ext3-fast), which retains much of the positive ordering and atomicity properties of ext3 in data journaling mode, without the high cost. Such verification would have been useful in the past; when delayed allocation was introduced in Linux ext4, it broke several applications, resulting in bug reports, extensive mailing-list discussions, widespread data loss, and finally, file-system changes [14]. With ALICE, testing the impact of changing persistence properties can become part of the file-system design process.

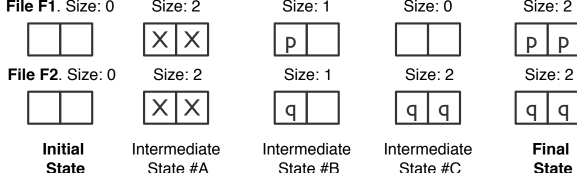


Figure 2: **Crash States.** *The figure shows the initial, final, and some of the intermediate crash states possible for the workload described in Section 2.1 . X represents garbage data in the files. Intermediate states #A and #B represent different kinds of atomicity violations, while intermediate state #C represents an ordering violation.*

2 Persistence Properties

In this section, we study the *persistence properties* of modern file systems. These properties determine which possible post-crash file system states are possible for a given file system; as we will see, different file systems provide subtly different guarantees, making the challenge of building correct application protocols atop such systems more vexing.

We begin with an example, and then describe our methodology: to explore possible on-disk states by reordering the I/O block stream, and then examine possible resulting states. Our testing is not complete, but finds persistence properties that do *not* hold for a file-system implementation. We then discuss our findings for six widely-used Linux file systems: ext2 [6], ext3 [51], ext4 [50], btrfs [30], xfs [45], and reiserfs [37].

Application-level crash consistency depends strongly upon these persistence properties, yet there are currently no standards. We believe that defining and studying persistence properties is the first step towards standardizing them across file systems.

2.1 An Example

All application update protocols boil down to a sequence of I/O-related system calls which modify on-disk state. Two broad properties of system calls affect how they are persisted. The first is *atomicity*: does the update from the call happen all at once, or are there possible intermediate states that might arise due to an untimely crash? The second is *ordering*: can this system call be persisted *after* a later system call? We now explain these properties with an example.

We consider the following pseudo-code snippet:

```
write(f1, "pp");
write(f2, "qq");
```

In this example, the application first appends the string `pp` to file descriptor `f1` and then appends the string `qq` to file descriptor `f2`. Note that we will sometimes refer to such a `write()` as an `append()` for simplicity.

Figure 2 shows a few possible crash states that can result. If the append is not *atomic*, for example, it would be possible for the *size* of the file to be updated without the new data reflected to disk; in this case, the files could contain garbage, as shown in State A in the diagram. We refer to this as *size-atomicity*. A lack of atomicity could also be realized with only part of a write reaching disk, as shown in State B. We refer to this as *content-atomicity*.

If the file system persists the calls out of order, another outcome is possible (State C). In this case, the second write reaches the disk first, and as a result only the second file is updated. Various combinations of these states are also possible.

As we will see when we study application update protocols, modern applications expect different atomicity and ordering properties from underlying file systems. We now study such properties in detail.

2.2 Study and Results

We study the persistence properties of six Linux file systems: ext2, ext3, ext4, btrfs, xfs, and reiserfs. A large number of applications have been written targeting these file systems. Many of these file systems also provide multiple configurations that make different trade-offs between performance and consistency: for instance, the data journaling mode of ext3 provides the highest level of consistency, but often results in poor performance [35]. Between file systems and their various configurations, it is challenging to know or reason about which persistence properties are provided. Therefore, we examine different configurations of the file systems we study (a total of 16).

To study persistence properties, we built a tool, known as the *Block Order Breaker* (BOB), to empirically find cases where various persistence properties do *not* hold for a given file system. BOB first runs a simple user-supplied workload designed to stress the persistence property tested (e.g., a number of writes of a specific size to test overwrite atomicity). BOB collects the block I/O generated by the workload, and then re-orders the collected blocks, selectively writing some of them to disk to generate a new *legal* disk state (disk barriers are obeyed). In this manner, BOB generates a number of unique disk images corresponding to possible on-disk states after a system crash. BOB then runs file-system recovery on each resulting disk image, and checks whether various persistence properties hold (e.g., if writes were atomic). If BOB finds even a single disk image where the checker fails, then we know that the property does not hold on the file system. Proving the converse (that a property holds in all situations) is not possible using BOB; currently, only simple block re-orderings and all prefixes of the block trace are tested.

Note that different system calls (e.g., `writev()`, `write()`) lead to the same file-system output. We group such calls together into a generic file-system update we term an *operation*. We have found that grouping all operations into three major categories is sufficient for our purposes here: file overwrite, file append, and directory operations (including rename, link, unlink, mkdir, etc.).

Persistence Property	File system															
	ext2	ext2-sync	ext3-writeback	ext3-ordered	ext3-datajournal	ext4-writeback	ext4-ordered	ext4-nodelalloc	ext4-datajournal	btrfs	xfs	xfs-wsync	reiserfs-nolog	reiserfs-writeback	reiserfs-ordered	reiserfs-datajournal
Atomicity																
Single sector overwrite																
Single sector append	×		×			×								×		
Single block overwrite	×	×	×	×		×	×	×			×	×	×	×	×	
Single block append	×	×	×			×							×	×		
Multi-block append/writes	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Multi-block prefix append	×	×	×			×							×	×		
Directory op	×	×											×			
Ordering																
Overwrite → Any op	×		×	×		×	×	×			×	×	×	×	×	
[Append, rename]→ Any op	×		×			×							×	×		
O_TRUNC Append → Any op	×		×			×							×	×		
Append → Append (same file)	×		×			×							×	×		
Append → Any op	×		×			×	×			×	×		×	×		
Dir op → Any op	×									×			×			

Table 1: **Persistence Properties.** *The table shows atomicity and ordering persistence properties that we empirically determined for different configurations of file systems. X → Y indicates that X is persisted before Y. [X,Y] → Z indicates that Y follows X in program order, and both become durable before Z. A × indicates that we have a reproducible test case where the property fails in that file system.*

Table 1 lists the results of our study. The table shows, for each file system (and specific configuration) whether a particular persistence property has been found to *not* hold; such cases are marked with an ×.

The size and alignment of an overwrite or append affects its atomicity. Hence, we show results for single sector, single block, and multi-block overwrite and append operations. For ordering, we show whether given properties hold assuming different orderings of overwrite, append, and directory operations; the append operation has some interesting special cases relating to delayed allocation (as found in Linux ext4) – we show these separately.

2.2.1 Atomicity

We observe that all tested file systems seemingly provide atomic single-sector overwrites: in some cases (e.g., ext3-ordered), this property arises because the underlying disk provides atomic sector writes. Note that if such file systems are run on top of new technologies (such as PCM) that provide only byte-level atomicity [12], single-sector overwrites will not be atomic.

Providing atomic appends requires the update of two locations (file inode, data block) atomically. Doing so requires file-system machinery, and is not provided by ext2 or writeback configurations of ext3, ext4, and reiserfs.

Overwriting an entire block atomically requires data journaling or copy-on-write techniques; atomically appending an entire block can be done using ordered mode journaling, since the file system only needs to ensure the entire block is persisted before adding a pointer to it.

Current file systems do not provide atomic multi-block appends; appends can be broken down into multiple operations. However, most file systems seemingly guarantee that some prefix of the data written (e.g., the first 10 blocks of a larger append) will be appended atomically.

Directory operations such as `rename()` and `link()` are seemingly atomic on all file systems that use techniques like journaling or copy-on-write for consistency.

2.2.2 Ordering

We observe that ext3, ext4, and reiserfs in data journaling mode, and ext2 in sync mode, persist all tested operations in order. Note that these modes often result in poor performance on many workloads [35].

The append operation has interesting special cases. On file systems with the delayed allocation feature, it may be persisted after other operations. A special exception to this rule is when a file is appended, and then renamed. Since this idiom is commonly used to atomically update files [14], many file systems recognize it and allocate blocks immediately. A similar special case is appending to files that have been opened with O_TRUNC. Even with delayed allocation, successive appends to the same file are persisted in order. Linux ext2 and btrfs freely reorder directory operations (especially operations on different directories [11]) to increase performance.

2.3 Summary

From Table 1, we observe that persistence properties vary *widely* among file systems, and even among different configurations of the same file system. The order of persistence of system calls depends upon small details like whether the calls are to the same file or whether the file was renamed. From the viewpoint of an application developer, it is risky to assume that any particular property will be supported by all file systems.

3 The Application-Level Intelligent Crash Explorer (ALICE)

We have now seen that file systems provide different persistence properties. However, some important questions remain: How do current applications update their on-disk structures? What do they assume about the underlying file systems? Do such update protocols have vulnerabilities? To address these questions, we developed ALICE *(Application-Level Intelligent Crash Explorer)*. ALICE constructs different on-disk file states that may result due to a crash, and then verifies application correctness on each created state.

Unlike other approaches [53, 54] that simply test an application atop a given storage stack, ALICE finds the generic persistence properties required for application correctness, without being restricted to only a specified

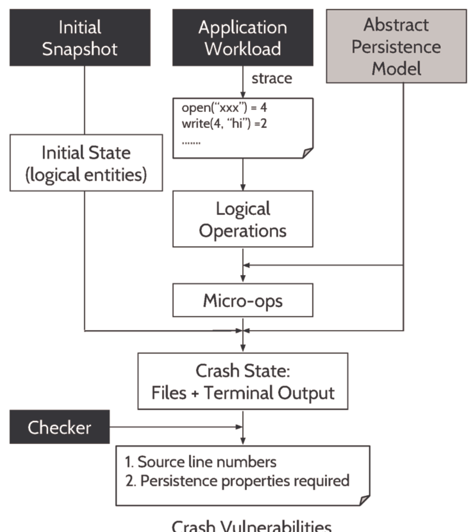


Figure 3: **ALICE Overview.** *The figure shows how ALICE converts user inputs into crash states and finally into crash vulnerabilities. Black boxes are user inputs. Grey boxes are optional inputs.*

file system. ALICE associates discovered vulnerabilities directly with source lines, and targets specific states that are prone to reveal crash vulnerabilities in different source lines. ALICE achieves this by constructing file states *directly* from the system-call trace of an application workload. The states to be explored and verified can be described purely in terms of system calls: the actual storage stack is not involved. ALICE can also be used to abstractly test the safety of new file systems.

We first describe how ALICE is used (§3.1). We then describe how ALICE calculates states possible during a system crash, using an Abstract Persistence Model (APM) (§3.2). Next, we describe how these states are selectively explored so as to discover application requirements in terms of persistence properties (§3.3), and how discovered vulnerabilities are reported associated with source code lines (§3.4). Finally, we describe our implementation (§3.5) and its limitations (§3.6).

3.1 Usage

ALICE is simple to use. The user first supplies ALICE with an initial snapshot of the files used by the application (typically an entire directory), and a workload script that exercises the application (such as performing a transaction). The user also supplies a checker script corresponding to the workload that verifies whether invariants of the workload are maintained (such as atomicity of the transaction). ALICE runs the checker atop different *crash states*, i.e., the state of files after rebooting from a system crash that can occur during the workload. ALICE then produces a logical representation of the update protocol executed during the workload, vulnerabilities in the protocol and their associated source lines, and persistence properties required for correctness.

```
# Workload

# Opening database
db = gdbm.open('mydb')

# Inserting key-value
db['x'] = 'foo'
db.sync()

print 'Done'
```

```
# Checker
db = gdbm.open('mydb')
c = len(db.keys())
if alice.printed('Done'):
  assert c == 1
else:
  assert c == 0 or c == 1
if c == 1:
  assert db['x'] == 'foo'
```

Listing 1: **Workload and Checker.** *Simplified form of python workload and checker for GDBM (a key-value store).*

The exact crash states possible for a workload varies with the file system. For example, depending on the file system, appending to a file can result in the file containing either a prefix of the data persisted, with random data intermixed with file data, or various combinations thereof. ALICE uses file-system *Abstract Persistence Models (APMs)* to define the exact crash states possible in a given file system. By default, ALICE uses an APM with few restrictions on the possible crash states, so as to find generic persistence properties required for application correctness. However, ALICE can be restricted to find vulnerabilities occurring only on a specific file system, by supplying the APM of that file system.

Listing 1 shows example workload and checker scripts for GDBM, a key-value store, written in Python. We discuss how APMs are specified in the next subsection.

3.2 Crash States and APMs

Figure 3 shows an overview of the steps ALICE follows to find crash vulnerabilities. The user-supplied workload is first run, and a system-call trace obtained; the trace represents an execution of the application's update protocol. The trace is converted into a sequence of *logical operations* by ALICE. The sequence of logical operations, along with an APM, is used to calculate the different crash states that are possible from the initial state. These steps are now explained in detail.

3.2.1 Logical Operations

ALICE first converts the trace of system calls in the application workload to *logical operations*. Logical operations abstract away details such as current read and write offsets, file descriptors, and transform a large set of system calls and other I/O producing behavior into a small set of file-system operations. For example, `write()`, `pwrite()`, `writev()`, `pwritev()`, and `mmap()`-writes are all translated into `overwrite` or `append` logical operations. Logical operations also associate a conceptual inode to each file or directory involved.

3.2.2 Abstract Persistence Models

An APM specifies *all* constraints on the atomicity and ordering of logical operations for a given file system, thus defining which crash states are possible.

APMs represent crash states as consisting of two logical entities: *file inodes* containing data and a file size,

Logical Operation	Micro-operations
overwrite	$N\times$ write_block(data)
append	$N\times$ { change_file_size; write_block(random); write_block(data) }
truncate	$N\times$ { change_file_size; write_block(random); write_block(zeroes) }
link	create_dir_entry
unlink	delete_dir_entry + *truncate if last link*
rename	delete_dir_entry(dest) + *truncate if last link* create_dir_entry(dest) delete_dir_entry(source)
print	stdout

(a) Atomicity Constraints.

Description	Constraint
sync-ops	[$any\text{-}op_i$(A) ... fsync$_j$(A)]→$any\text{-}op_k$ $\forall\, i < j < k$
stdout	stdout$_i$()→$any\text{-}op_j$ $\forall\, i < j$

(b) Ordering Constraints.

Table 2: **Default APM Constraints.** *(a) shows atomicity constraints; N indicates a logical operation being divided into many micro-ops. (b) shows ordering constraints. X_i is the ith operation, and any-op(A) is an operation on the file or directory A.*

and *directories* containing directory entries. Each logical operation operates on one or more of these entities. An infinite number of instances of each logical entity exist, and they are never allocated or de-allocated, but rather simply changed. Additionally, each crash state also includes any output printed to the terminal during the time of the crash as a separate entity.

To capture intermediate crash states, APMs break logical operations into *micro-operations*, i.e., the smallest atomic modification that can be performed upon each logical entity. There are five micro-ops:

- *write_block*: A write of size *block* to a file. Two special arguments to *write_block* are *zeroes* and *random*: *zeroes* indicates the file system initializing a newly allocated block to zero; *random* indicates an uninitialized block. Writes beyond the end of a file cause data to be stored without changing file size.
- *change_file_size*: Changes the size of a file inode.
- *create_dir_entry*: Creates a directory entry in a directory, and associates a file inode or directory with it.
- *delete_dir_entry*: Deletes a directory entry.
- *stdout*: Adds messages to the terminal output.

The APM specifies atomicity constraints by defining how logical operations are translated into micro-ops. The APM specifies ordering constraints by defining which micro-ops can reach the disk before other micro-ops.

In most cases, we utilize a default APM to find the greatest number of vulnerabilities in application update protocols. The atomicity constraints followed by this default file system are shown in Table 2(a), which specifically shows how each logical operation is broken down into micro-ops. The ordering constraints imposed by the default APM are quite simple, as seen in Table 2(b): all micro-ops followed by a sync on a file A are ordered after

```
open(path="/x2VC") = 10
Micro-ops: None
Ordered after: None

pwrite(fd=10, offset=0, size=1024)
Micro-ops: #1 write_block(inode=8, offset=0, size=512)
           #2 write_block(inode=8, offset=512, size=512)
Ordered after: None

fsync(10)
Micro-ops: None
Ordered after: None

pwrite(fd=10, offset=1024, size=1024)
Micro-ops: #3 write_block(inode=8, offset=1024, size=512)
           #4 write_block(inode=8, offset=1536, size=512)
Ordered after: #1, #2

link(oldpath="/x2VC", newpath="/file")
Micro-ops: #5 create_dir_entry(dir=2, entry='file', inode=8)
Ordered after: #1, #2

write(fd=1, data="Writes recorded", size=15)
Micro-ops: #6 stdout("Writes recorded")
Ordered after: #1, #2
```

Listing 2: **Annotated Update Protocol Example.** *Micro-operations generated for each system call are shown along with their dependencies. The inode number of x2VC is 8, and for the root directory is 2. Some details of listed system calls have been omitted.*

writes to A that precede the sync. Similar ordering also applies to *stdout*, and additionally, all operations following an *stdout* must be ordered after it.

ALICE can also model the behavior of real file systems when configured with other APMs. As an example, for the ext3 file system under the *data=journal* mode, the ordering constraint is simply that each micro-op depends on all previous micro-ops. Atomicity constraints for ext3 are mostly simple: all operations are atomic, except file writes and truncates, which are split at block-granularity. Atomic renames are imposed by a circular ordering dependency between the micro-ops of each rename.

3.2.3 Constructing crash states.

As explained, using the APM, ALICE can translate the system-call trace into micro-ops and calculate ordering dependencies amongst them. Listing 2 shows an example system-call trace, and the resulting micro-ops and ordering constraints. ALICE also represents the initial snapshot of files used by the application as logical entities.

ALICE then selects different sets of the translated micro-ops that obey the ordering constraints. A new crash state is constructed by sequentially applying the micro-ops in a selected set to the initial state (represented as logical entities). For each crash state, ALICE then converts the logical entities back into actual files, and supplies them to the checker. The user-supplied checker thus verifies the crash state.

3.3 Finding Application Requirements

By default, ALICE targets specific crash states that concern the ordering and atomicity of each individual system call. The explored states thus relate to basic persistence properties like those discussed in Section 2, making it

straightforward to determine application requirements. We now briefly describe the crash states explored.

Atomicity *across* System Calls. The application update protocol may require multiple system calls to be persisted together atomically. This property is easy to check: if the protocol has N system calls, ALICE constructs one crash state for each prefix (i.e., the first X system calls, $\forall\ 1 < X < N$) applied. In the sequence of crash states generated in this manner, the first crash state to have an application invariant violated indicates the start of an atomic group. The invariant will hold once again in crash states where all the system calls in the atomic group are applied. If ALICE determines that a system call X is part of an atomic group, it does not test whether the protocol is vulnerable to X being persisted out of order, or being partially persisted.

System-Call Atomicity. The protocol may require a single system call to be persisted atomically. ALICE tests this for each system call by applying all previous system calls to the crash state, and then generating crash states corresponding to different intermediate states of the system call and checking if application invariants are violated. The intermediate states for file-system operations depend upon the APM, as shown (for example) in Table 2. Some interesting cases include how ALICE handles appends and how it explores the atomicity of writes. For appends, we introduce intermediate states where blocks are filled with random data; this models the update of the size of a file reaching disk before the data has been written. We split overwrites and appends in two ways: into block-sized micro-operations, and into three parts regardless of size. Though not exhaustive, we have found our exploration of append and write atomicity useful in finding application vulnerabilities.

Ordering Dependency among System Calls. The protocol requires system call A to be persisted before B if a crash state with B applied (and not A) violates application invariants. ALICE tests this for each pair of system calls in the update protocol by applying every system call from the beginning of the protocol until B except for A.

3.4 Static Vulnerabilities

ALICE must be careful in how it associates problems found in a system-call trace with source code. For example, consider an application issuing ten writes in a loop. The update protocol would then contain ten `write()` system calls. If each write is required to be atomic for application correctness, ALICE detects that *each* system call is involved in a vulnerability; we term these as **dynamic** vulnerabilities. However, the cause of all these vulnerabilities is a single source line. ALICE uses stack trace information to correlate all 10 system calls to the line, and reports it as a single **static** vulnerability. In the rest of this paper, we only discuss static vulnerabilities.

3.5 Implementation

ALICE consists of around 4000 lines of Python code, and also traces memory-mapped writes in addition to system calls. It employs a number of optimizations.

First, ALICE caches crash states, and constructs a new crash state by incrementally applying micro-operations onto a cached crash state. We also found that the time required to check a crash state was much higher than the time required to incrementally construct a crash state. Hence, ALICE constructs crash states sequentially, but invokes checkers concurrently in multiple threads.

Different micro-op sequences can lead to the same crash state. For example, different micro-op sequences may write to different parts of a file, but if the file is unlinked at the end of sequence, the resulting disk state is the same. Therefore, ALICE hashes crash states and only checks the crash state if it is new.

We found that many applications write to debug logs and other files that do not affect application invariants. ALICE filters out system calls involved with these files.

3.6 Limitations

ALICE is not complete, in that there may be vulnerabilities that are not detected by ALICE. It also requires the user to write application workloads and checkers; we believe workload automation is orthogonal to the goal of ALICE, and various model-checking techniques can be used to augment ALICE. For workloads that use multiple threads to interact with the file system, ALICE serializes system calls in the order they were issued; in most cases, this does not affect vulnerabilities as the application uses some form of locking to synchronize between threads. ALICE currently does not handle file attributes; it would be straight-forward to extend ALICE to do so.

4 Application Vulnerabilities

We study 11 widely used applications to find whether file-system behavior significantly affects application users, which file-system behaviors are thus important, and whether testing using ALICE is worthwhile in general. One of ALICE's unique advantages, of being able to find targeted vulnerabilities under an abstract file system and reporting them in terms of a persistence property violated, is thus integral to the study. The applications each represent different domains, and range in maturity from a few years-old to decades-old. We study three key-value stores (LevelDB [20], GDBM [19], LMDB [46]), three relational databases (SQLite [43], PostgreSQL [49], HSQLDB [23]), two version control systems (Git [26], Mercurial [31]), two distributed systems (HDFS [40], ZooKeeper [3]), and a virtualization software (VMWare Player [52]). We study two versions of LevelDB (1.10, 1.15), since they vary considerably in their update-protocol implementation.

Aiming towards the stated goal of the study, we try to consider typical user expectations and deployment scenarios for applications, rather than only the guarantees listed in their documentation. Indeed, for some applications (Git, Mercurial), we could not find any documented guarantees. We also consider file-system behaviors that may not be common now, but may become prevalent in the future (especially with new classes of I/O devices). Moreover, the number of vulnerabilities we report (in each application) only relates to the number of source code lines depending on file-system behavior. Note that, due to these reasons, the study is not suitable for comparing the correctness between different applications, or towards strictly verifying application correctness.

We first describe the workloads and checkers used in detecting vulnerabilities (§4.1). We then present an overview of the protocols and vulnerabilities found in different applications (§4.2). We discuss the importance of the discovered vulnerabilities (§4.3), interesting patterns observable among the vulnerabilities (§4.4), and whether vulnerabilities are exposed on current file systems (§4.5). We also evaluate whether ALICE can validate new file-system designs (§4.6).

4.1 Workloads and Checkers

Most applications have configuration options that change the update protocol or application crash guarantees. Our workloads test a total of 34 such configuration options across the 11 applications. Our checkers are conceptually simple: they do read operations to verify workload invariants for that particular configuration, and then try writes to the datastore. However, some applications have complex invariants, and recovery procedures that they expect users to carry out (such as removing a leftover lock file). Our checkers are hence complex (e.g., about 500 LOC for Git), invoking all recovery procedures we are aware of that are expected of normal users.

We now discuss the workloads and checkers for each application class. Where applicable, we also present the guarantees we believe each application makes to users, information garnered from documentation, mailing-list discussions, interaction with developers, and other relevant sources.

Key-value Stores and Relational Databases. Each workload tests different parts of the protocol, typically opening a database, and inserting enough data to trigger checkpoints. The checkers check for atomicity, ordering, and durability of transactions. We note here that GDBM does not provide any crash guarantees, though we believe lay users will be affected by any loss of integrity. Similarly, SQLite does not provide durability under the default journal-mode (we became aware of this only after interacting with developers), but its documentation seems misleading. We enable checksums on LevelDB.

Version Control Systems. Git's crash guarantees are fuzzy; mailing-list discussions suggest that Git expects a fully-ordered file system [27]. Mercurial does not provide *any* guarantees, but does provide a plethora of manual recovery techniques. Our workloads add two files to the repository and then commit them. The checker uses commands like `git-log`, `git-fsck`, and `git-commit` to verify repository state, checking the integrity of the repository and the durability of the workload commands. The checkers remove any leftover lock files, and perform recovery techniques that do not discard committed data or require previous backups.

Virtualization and Distributed Systems. The VMWare Player workload issues writes and flushes from within the guest; the checker repairs the virtual disk and verifies that flushed writes are durable. HDFS is configured with replicated metadata and restore enabled. HDFS and ZooKeeper workloads create a new directory hierarchy; the checker tests that files created before the crash exist. In ZooKeeper, the checker also verifies that quota and ACL modifications are consistent.

If ALICE finds a vulnerability related to a system call, it does not search for other vulnerabilities related to the same call. If the system call is involved in multiple, logically separate vulnerabilities, this has the effect of hiding some of the vulnerabilities. Most tested applications, however, have distinct, independent sets of failures (e.g., *dirstate* and *repository* corruption in Mercurial, consistency and durability violation in other applications). We use different checkers for each type of failure, and report vulnerabilities for each checker separately.

Summary. If application invariants for the tested configuration are explicitly and conspicuously documented, we consider violating those invariants as failure; otherwise, our checkers consider violating a lay user's expectations as failure. We are careful about any recovery procedures that need to be followed on a system crash. Space constraints here limit exact descriptions of the checkers; we provide more details in our webpage [2].

4.2 Overview

We now discuss the logical protocols of the applications examined. Figure 4 visually represents the update protocols, showing the logical operations in the protocol (organized as modules) and discovered vulnerabilities.

4.2.1 Databases and Key-Value Stores

Most databases use a variant of write-ahead logging. First, the new data is written to a log. Then, the log is checkpointed or compacted, i.e., the actual database is usually overwritten, and the log is deleted.

Figure 4(A) shows the protocol used by LevelDB-1.15. LevelDB adds inserted key-value pairs to the log until it reaches a threshold, and then switches to a new

Figure 4: **Protocol Diagrams.** *The diagram shows the modularized update protocol for all applications. For applications with more than one configuration (or versions), only a single configuration is shown (SQLite: Rollback, LevelDB: 1.15). Uninteresting parts of the protocol and a few vulnerabilities (similar to those already shown) are omitted. Repeated operations in a protocol are shown as 'N ×' next to the operation, and portions of the protocol executed conditionally are shown as '? ×'. Blue-colored text simply highlights such annotations and sync calls. Ordering and durability dependencies are indicated with arrows, and dependencies between modules are indicated by the numbers on the arrows, corresponding to line numbers in modules. Durability dependency arrows end in an* stdout *micro-op; additionally, the two dependencies marked with * in HSQLDB are also durability dependencies. Dotted arrows correspond to* safe rename *or* safe file flush *vulnerabilities discussed in Section 4.4. Operations inside brackets must be persisted together atomically.*

log; during the switch, a background thread starts compacting the old log file. Figure 4(A)(i) shows the compaction; Figure 4(A)(ii) shows the appends to the log file. During compaction, LevelDB first writes data to a new *ldb* file, updates pointers to point to the new file (by appending to a *manifest*), and then deletes the old log file.

In LevelDB, we find vulnerabilities occurring while appending to the log file. A crash can result in the appended portion of the file containing garbage; LevelDB's recovery code does not properly handle this situation, and the user gets an error if trying to access the inserted key-value pair (which should not exist in the database). We also find some vulnerabilities occurring during compaction. For example, LevelDB does not explicitly persist the directory entries of *ldb* files; a crash might cause the files to vanish, resulting in unavailability.

Some databases follow protocols that are radically different from write-ahead logging. For example, LMDB uses shadow-paging (copy-on-write). LMDB requires that the final pointer update (106 bytes) in the copy-on-write tree to be atomic. HSQLDB uses a combination of write-ahead logging and update-via-rename, on the same files, to maintain consistency. The update-via-rename is performed by first separately unlinking the destination file, and then renaming; out-of-order persistence of `rename()`, `unlink()`, or log creation causes problems.

4.2.2 Version Control Systems

Git and Mercurial maintain meta-information about their repository in the form of logs. The Git protocol is illustrated in Figure 4(G). Git stores information in the form of object files, which are never modified; they are created as temporary files, and then linked to their permanent file names. Git also maintains pointers in separate files, which point to both the meta-information log and the object files, and are updated using update-via-rename. Mercurial, on the other hand, uses a journal to maintain consistency, using update-via-rename only for a few unimportant pieces of information.

We find many ordering dependencies in the Git protocol, as shown in Figure 4(G). This result is not surprising, since mailing-list discussions suggest Git developers expect total ordering from the file system. We also find a Git vulnerability involving atomicity across multiple system calls; a pointer file being updated (via an append) has to be persisted atomically with another file getting updated (via an update-via-rename). In Mercurial, we find many ordering vulnerabilities for the same reason, not being designed to tolerate out-of-order persistence.

4.2.3 Virtualization and Distributed Systems

VMWare Player's protocol is simple. VMWare maintains a static, constant mapping between blocks in the virtual disk, and in the `VMDK` file (even for dynamically allocated `VMDK` files); directly overwriting the `VMDK` file maintains consistency (though VMWare does use update-via-rename for some small files). Both HDFS and ZooKeeper use write-ahead logging. Figure 4(K) shows the ZooKeeper logging module. We find that ZooKeeper does not explicitly persist directory entries of log files, which can lead to lost data. ZooKeeper also requires some log writes to be atomic.

4.3 Vulnerabilities Found

ALICE finds 60 static vulnerabilities in total, corresponding to 156 dynamic vulnerabilities. Altogether, applications failed in more than 4000 crash states. Table 3(a) shows the vulnerabilities classified by the affected persistence property, and 3(b) shows the vulnerabilities classified by failure consequence. Table 3(b) also separates out those vulnerabilities related only to user expectations and not to documented guarantees, with an asterik (*); many of these correspond to applications for which we could not find any documentation of guarantees.

The different journal-mode configurations provided by SQLite use different protocols, and the different versions of LevelDB differ on whether their protocols are designed around the `mmap()` interface. Tables 3(a) and 3(b) hence show these configurations of SQLite and LevelDB separately. All other configurations (in all applications) do not change the basic protocol, but vary on the application invariants; among different configurations of the same update protocol, all vulnerabilities are revealed in the *safest* configuration. Table 3 and the rest of the paper only show vulnerabilities we find in the safest configuration, i.e., we do not count separately the same vulnerabilities from different configurations of the same protocol.

We find many vulnerabilities have severe consequences such as silent errors or data loss. Seven applications are affected by data loss, while two (both LevelDB versions and HSQLDB) are affected by silent errors. The *cannot open* failures include failure to start the server in HDFS and ZooKeeper, while the *failed reads and writes* include basic commands (e.g., `git-log`, `git-commit`) failing in Git and Mercurial. A few *cannot open* failures and *failed reads and writes* might be solvable by application experts, but we believe lay users would have difficulty recovering from such failures (our checkers invoke standard recovery techniques). We also checked if any discovered vulnerabilities are previously known, or considered inconsequential. The single PostgreSQL vulnerability is documented; it can be solved with nonstandard (although simple) recovery techniques. The single LMDB vulnerability is discussed in a mailing list, though there is no available workaround. All these previously known vulnerabilities are separated out in Table 3(b) (†). The five *dirstate fail* vulnerabilities in Mercurial are shown separately, since they are less harmful than other vulnerabilities (though frustrating to the lay

Application	Types: Across-syscalls atomicity	Atomicity: Appends and truncates	Atomicity: Single-block overwrites	Atomicity: Renames and unlinks	Ordering: Safe file flush	Ordering: Safe renames	Ordering: Other	Durability: Safe file flush	Durability: Other	Unique static vulnerabilities
Leveldb1.10	1‡	1		1	2	1	3	1		**10**
Leveldb1.15	1	1		1	1		2			**6**
LMDB			1							**1**
GDBM	1			1			1		2	**5**
HSQLDB		1		2	1		3	2	1	**10**
Sqlite-Roll									1	**1**
Sqlite-WAL										**0**
PostgreSQL			1							**1**
Git	1			1	2	1	3		1	**9**
Mercurial	2	1		1		1	4		2	**10**
VMWare				1						**1**
HDFS				1			1			**2**
ZooKeeper			1				1	2		**4**
Total	**6**	**4**	**3**	**9**	**6**	**3**	**18**	**5**	**7**	**60**

(a) Types.

Application	Silent errors	Data loss	Cannot open	Failed reads and writes	Other
Leveldb1.10	1	1	5	4	
Leveldb1.15	2		2	2	
LMDB					read-only open†
GDBM		2*	3*		
HSQLDB	2	3	5		
Sqlite-Roll		1*			
Sqlite-WAL					
PostgreSQL			1†		
Git		1*	3*	5*	3#*
Mercurial		2*	1*	6*	5 dirstate fail*
VMWare			1*		
HDFS			2*		
ZooKeeper		2*	2*		
Total	**5**	**12**	**25**	**17**	**9**

(b) Failure Consequences.

Application	ext3-w	ext3-o	ext3-j	ext4-o	btrfs
Leveldb1.10	3	1	1	2	4
Leveldb1.15	2	1	1	2	3
LMDB					
GDBM	3	3	2	3	4
HSQLDB					4
Sqlite-Roll	1	1	1	1	1
Sqlite-WAL					
PostgreSQL					
Git	2	2	2	2	5
Mercurial	4	3	3	6	8
VMWare					
HDFS					1
ZooKeeper	1	1		1	1
Total	16	12	10	17	31

(c) Under Current File Systems.

	Ordering	DO	AG	CA
ext3-w	Dir ops and file-sizes ordered among themselves, before sync operations.	✓	4K	×
ext3-o	Dir ops, appends, truncates ordered among themselves. Overwrites before non-overwrites, all before sync.	✓	4K	✓
ext3-j	All operations are ordered.	✓	4K	✓
ext4-o	Safe rename, safe file flush, dir ops ordered among themselves	✓	4K	✓
btrfs	Safe rename, safe file flush	✓	4K	✓

(d) APMs considered.

Table 3: **Vulnerabilities.** *(a) shows the discovered static vulnerabilities categorized by the type of persistence property. The number of unique vulnerabilities for an application can be different from the sum of the categorized vulnerabilities, since the same source code lines can exhibit different behavior. ‡ The atomicity vulnerability in Leveldb1.10 corresponds to multiple mmap() writes. (b) shows the number of static vulnerabilities resulting in each type of failure. † Previously known failures, documented or discussed in mailing lists. * Vulnerabilities relating to unclear documentation or typical user expectations beyond application guarantees. # There are 2 fsck-only and 1 reflog-only errors in Git. (c) shows the number of vulnerabilities that occur on current file systems (all applications are vulnerable under future file systems). (d) shows APMs used for calculating Table (c). Legend: DO: directory operations atomicity. AG: granularity of size-atomicity. CA: Content-Atomicity.*

user). Git's *fsck-only* and *reflog-only* errors are potentially dangerous, but do not seem to affect normal usage.

We interacted with the developers of eight applications, reporting a subset of the vulnerabilities we find. Our interactions convince us that the vulnerabilities will affect users if they are exposed. The other applications (GDBM, Git, and Mercurial) were not designed to provide crash guarantees, although we believe their users will be affected by the vulnerabilities found should an untimely crash occur. Thus, the vulnerabilities will not surprise a developer of these applications, and we did not report them. We also did not report vulnerabilities concerning partial renames (usually dismissed since they are not commonly exposed), or documented vulnerabilities.

Developers have acted on five of the vulnerabilities we find: one (LevelDB-1.10) is now fixed, another (LevelDB-1.15) was fixed parallel to our discovery, and three (HDFS, and two in ZooKeeper) are under consideration. We have found that developers often dismiss other vulnerabilities which do not (or are widely believed to not) get exposed in current file systems, especially relating to out-of-order persistence of directory operations. The fact that only certain operating systems allow an `fsync()` on a directory is frequently referred to; both HDFS and ZooKeeper respondents lament that such an `fsync()` is not easily achievable with Java. The developers suggest the SQLite vulnerability is actually not a behavior guaranteed by SQLite (specifically, that durability cannot be achieved under `rollback` journaling); we believe the documentation is misleading.

Of the five acted-on vulnerabilities, three relate to not explicitly issuing an `fsync()` on the parent directory after creating and calling `fsync()` on a file. However, not issuing such an `fsync()` is perhaps more safe in modern file systems than out-of-order persistence of directory operations. We believe the developers' interest in fixing this problem arises from the Linux documentation explicitly recommending an `fsync()` after creating a file.

Summary. ALICE detects 60 vulnerabilities in total, with 5 resulting in silent failures, 12 in loss of durability, 25 leading to inaccessible applications, and 17 returning errors while accessing certain data. ALICE is also able to detect previously known vulnerabilities.

4.4 Common Patterns

We now examine vulnerabilities related with different persistence properties. Since durability vulnerabilities show a separate pattern, we consider them separately.

4.4.1 Atomicity across System Calls

Four applications (including both versions of LevelDB) require atomicity across system calls. For three applications, the consequences seem minor: inaccessibility during database creation in GDBM, dirstate corruption in Mercurial, and an erratic `reflog` in Git. LevelDB's vul-

nerability has a non-minor consequence, but was fixed immediately after introducing LevelDB-1.15 (when LevelDB started using `read()`-`write()` instead of `mmap()`).

In general, we observe that this class of vulnerabilities seems to affect applications less than other classes. This result may arise because these vulnerabilities are easily tested: they are exposed independent of the file system (i.e, via process crashes), and are easier to reproduce.

4.4.2 Atomicity within System Calls

Append atomicity. Surprisingly, three applications require appends to be *content-atomic*: the appended portion should contain actual data. The failure consequences are severe, such as corrupted reads (HSQLDB), failed reads (LevelDB-1.15) and repository corruption (Mercurial). Filling the appended portion with zeros instead of garbage still causes failure; only the current implementation of delayed allocation (where file size does not increase until actual content is persisted) works. Most appends seemingly do not need to be *block-atomic*; only Mercurial is affected, and the affected append also requires content-atomicity.

Overwrite atomicity. LMDB, PostgreSQL, and ZooKeeper require small writes ($<$ 200 bytes) to be atomic. Both the LMDB and PostgreSQL vulnerabilities are previously known.

We do not find any multi-block overwrite vulnerabilities, and even single-block overwrite requirements are typically documented. This finding is in stark contrast with append atomicity; some of the difference can be attributed to the default APM (overwrites are content-atomic), and to some workloads simply not using overwrites. However, the major cause seems to be the basic mechanism behind application update protocols: modifications are first logged, in some form, via appends; logged data is then used to overwrite the actual data. Applications have careful mechanisms to detect and repair failures in the actual data, but overlook the presence of garbage content in the log.

Directory operation atomicity. Given that most file systems provide atomic directory operations (§2.2), one would expect that most applications would be vulnerable to such operations not being atomic. However, we do not find this to be the case for certain classes of applications. Databases and key-value stores do not employ atomic renames extensively; consequently, we observe non-atomic renames affecting only three of these applications (GDBM, HSQLDB, LevelDB). Non-atomic unlinks seemingly affect only HSQLDB (which uses unlinks for logically performing renames), and we did not find any application affected by non-atomic truncates.

4.4.3 Ordering between System Calls

Applications are extremely vulnerable to system calls being persisted out of order; we find 27 vulnerabilities.

Safe renames. On file systems with delayed allocation, a common heuristic to prevent data loss is to persist all data (including appends and truncates) of a file before subsequent renames of the file [14]. We find that this heuristic only matches (and thus fixes) three discovered vulnerabilities, one each in Git, Mercurial, and LevelDB-1.10. A related heuristic, where updating *existing* files by opening them with `O_TRUNC` flushes the updated data while issuing a `close()`, does not affect any of the vulnerabilities we discovered. Also, the effect of the heuristics varies with minor details: if the safe-rename heuristic does not persist file truncates, only two vulnerabilities will be fixed; if the `O_TRUNC` heuristic also acts on new files, an additional vulnerability will be fixed.

Safe file flush. An `fsync()` on a file does not guarantee that the file's directory entry is also persisted. Most file systems, however, persist directory entries that the file is dependent on (e,g., directory entries of the file and its parent). We found that this behavior is required by three applications for maintaining basic consistency.

4.4.4 Durability

We find vulnerabilities in seven applications resulting in durability loss. Of these, only two applications (GDBM and Mercurial) are affected because an `fsync()` is not called on a file. Six applications require `fsync()` calls on directories: three are affected by *safe file flush* discussed previously, while four (HSQLDB, SQLite, Git, and Mercurial) require other `fsync()` calls on directories. As a special case, with HSQLDB, previously committed data is lost, rather than data that was being committed during the time of the workload. In all, only four out of the twelve vulnerabilities are exposed when full ordering is promised: many applications do issue an `fsync()` call before durability is essential, but do not `fsync()` all the required information.

4.4.5 Summary

We believe our study offers several insights for file-system designers. Future file systems should consider providing ordering between system calls, and atomicity within a system call in specific cases. Vulnerabilities involving atomicity of multiple system calls seem to have minor consequences. Requiring applications to separately flush the directory entry of a created and flushed file can often result in application failures. For durability, most applications seem to explicitly flush some, but not all, of the required information; thus, providing ordering among system calls can also help durability.

4.5 Impact on Current File Systems

Our study thus far has utilized an abstract (and weak) file system model (i.e., APM) in order to discover the broadest number of vulnerabilities. We now utilize specific file-system APMs to understand how modern protocols

would function atop a range of modern file systems and configurations. Specifically, we focus on Linux ext3 (including writeback, ordered, and data-journaling mode), Linux ext4, and btrfs. The considered APMs are based on our understanding of file systems from Section 2.

Table 3(c) shows the vulnerabilities reported by ALICE, while 3(d) shows the considered APMs. We make a number of observations based on Table 3(c). First, a significant number of vulnerabilities are exposed on *all* examined file systems. Second, ext3 with journaled data is the safest: the only vulnerabilities exposed relate to atomicity across system calls, and a few durability vulnerabilities. Third, a large number of vulnerabilities are exposed on btrfs as it aggressively persists operations out of order [11]. Fourth, some applications show no vulnerabilities on any considered APM; thus, the flaws we found in such applications do not manifest on today's file systems (but may do so on future systems).

Summary. Application vulnerabilities are exposed on many current file systems. The vulnerabilities exposed vary based on the file system, and thus testing applications on only a few file systems does not work.

4.6 Evaluating New File-System Designs

File-system modifications for improving performance have introduced wide-spread data loss in the past [14], because of changes to the file-system persistence properties. ALICE can be used to test whether such modifications break correctness of existing applications. We now describe how we use ALICE to evaluate a hypothetical variant of ext3 (data-journaling mode), *ext3-fast*.

Our study shows that ext3 (data-journaling mode) is the safest file system; however, it offers poor performance for many workloads [35]. Specifically, `fsync()` latency is extremely high as ext3 persists *all* previous operations on `fsync()`. One way to reduce `fsync()` latency would be to modify ext3 to persist only the synced file. However, other file systems (e.g,. btrfs) that have attempted to reduce `fsync()` latency [13] have resulted in increased vulnerabilities. Our study suggests a way to reduce latency *without* exposing more vulnerabilities.

Based on our study, we hypothesize that data that is not *synced* need not be persisted before explicitly synced data for correctness; such data must only be persisted in-order amongst itself. We design *ext3-fast* to reflect this: `fsync()` on a file A persists only A, while other dirty data and files are still persisted in-order.

We modeled *ext3-fast* in ALICE by slightly changing the APM of ext3 data journaling mode, so that synced directories, files, and their data, depend only on previous syncs and operations necessary for the file to exist (i.e., safe file flush is obeyed). The operations on a synced file are also ordered among themselves.

We test our hypothesis with ALICE; the observed ordering vulnerabilities (of the studied applications) are not exposed under *ext3-fast*. The design was not meant to fix durability or atomicity across system calls vulnerabilities, so those vulnerabilities are still reported by ALICE.

We estimate the performance gain of *ext3-fast* using the following experiment: we first write 250 MB to file A, then append a byte to file B and `fsync()` B. When both files are on the same ext3-ordered file system, `fsync()` takes about four seconds. If the files belong to different partitions on the same disk, mimicking the behavior of *ext3-fast*, the `fsync()` takes only 40 ms. The first case is 100 times slower because 250 MB of data is ordered before the single byte that needs to be persistent.

Summary. The *ext3-fast* file system (derived from inferences provided by ALICE) seems interesting for application safety, though further investigation is required into the validity of its design. We believe that the ease of use offered by ALICE will allow it to be incorporated into the design process of new file systems.

4.7 Discussion

We now consider why crash vulnerabilities occur commonly even among widely used applications. We find that application update protocols are complex and hard to isolate and understand. Many protocols are layered and spread over multiple files. Modules are also associated with other complex functionality (e.g., ensuring thread isolation). This complexity leads to issues that are obvious with a bird's eye view of the protocol: for example, HSQLDB's protocol has 3 consecutive `fsync()` calls to the same file (increasing latency). ALICE helps solve this problem by making it easy to obtain logical representations of update protocols as shown in Figure 4.

Another factor contributing to crash vulnerabilities is poorly written, untested recovery code. In LevelDB, we find vulnerabilities that should be prevented by correct implementations of the documented update protocols. Some recovery code is non-optimal: potentially recoverable data is lost in several applications (e.g., HSQLDB, Git). Mercurial and LevelDB provide utilities to verify or recover application data; we find these utilities hard to configure and error-prone. For example, an user invoking LevelDB's recovery command can unintentionally end up *further* corrupting the datastore, and be affected by (seemingly) unrelated configuration options (*paranoid checksums*). We believe these problems are a direct consequence of the recovery code being infrequently executed and insufficiently tested. With ALICE, recovery code can be tested on many corner cases.

Convincing developers about crash vulnerabilities is sometimes hard: there is a general mistrust surrounding such bug reports. Usually, developers are suspicious that the underlying storage stack might not respect `fsync()` calls [36], or that the drive might be corrupt. We hence

believe that most vulnerabilities that occur in the wild are associated with an incorrect root cause, or go unreported. ALICE can be used to easily reproduce vulnerabilities.

Unclear documentation of application guarantees contributes to the confusion about crash vulnerabilities. During discussions with developers about durability vulnerabilities, we found that SQLite, which proclaims itself as fully ACID-complaint, does not provide durability (even optionally) with the default storage engine, though the documentation suggests it does. Similarly, GDBM's GDBM_SYNC flag *does not* ensure durability. Users can employ ALICE to determine guarantees directly from the code, bypassing the problem of bad documentation.

5 Related Work

Our previous workshop paper [47] identifies the problem of application-level consistency depending upon file-system behavior, but is limited to two applications and does not use automated testing frameworks. Since we use ALICE to obtain results, our current study includes a greater number and variety of applications.

This paper adapts ideas from past work on dynamic program analysis and model checking. EXPLODE [53] has a similar flavor to our work: the authors use *in-situ* model checking to find crash vulnerabilities on different storage stacks. ALICE differs from EXPLODE in four significant ways. First, EXPLODE requires the target storage stack to be fully implemented; ALICE only requires a model of the target storage stack, and can therefore be used to evaluate application-level consistency on top of proposed storage stacks, while they are still at the design stage. Second, EXPLODE requires the user to carefully annotate complex file systems using *choose()* calls; ALICE requires the user to only specify a high-level APM. Third, EXPLODE reconstructs crash states by tracking I/O as it moves from the application to the storage. Although it is possible to use EXPLODE to determine the root cause of a vulnerability, we believe it is easier to do so using ALICE since ALICE checks for violation of specific persistence properties. Fourth, EXPLODE stops at finding crash vulnerabilities; by helping produce protocol diagrams, ALICE contributes to understanding the protocol itself. Like BOB, EXPLODE can be used to test persistence properties; however, while BOB only re-orders block I/O, EXPLODE can test re-orderings caused at different layers in the storage stack.

Zheng et al. [54] find crash vulnerabilities in databases. They contribute a standard set of workloads that stress databases (particularly, with multiple threads), and check ACID properties; the workloads and checkers can be used with ALICE. Unlike our work, Zheng et al. do not systematically explore vulnerabilities of each system call; they are limited by the re-orderings and non-atomicity exhibited by a particular (implemented) file system during a single workload execution. Thus, their work is more suited for finding those vulnerabilities that are *commonly* exposed under a given file system.

Woodpecker [15] can be used to find crash vulnerabilities when supplied with suspicious source code patterns to guide symbolic execution. Our work is fundamentally different to this approach, as ALICE does not require prior knowledge of patterns in checked applications.

Our work is influenced by SQLite's internal testing tool [43]. The tool works at an internal wrapper layer within SQLite, and is not helpful for generic testing.

RACEPRO [25], a testing tool for concurrency bugs, records system calls and replays them by splitting them into small operations, but does not test crash consistency.

OptFS [9], Featherstitch [16], and transactional file systems [17, 39, 42], discuss new file-system interfaces that will affect vulnerabilities. Our study can help inform the design of new interfaces by providing clear insights into what is missing in today's interfaces.

6 Conclusion

In this paper, we show how application-level consistency is dangerously dependent upon file system *persistence properties*, i.e., how file systems persist system calls. We develop BOB, a tool to test persistence properties and show that such properties vary widely among file systems. We build ALICE, a framework that analyzes application-level protocols and detects crash vulnerabilities. We analyze 11 applications, and find 60 vulnerabilities, some of which result in severe consequences like corruption or data loss. We present several insights derived from our study. The ALICE tool, and detailed descriptions of the vulnerabilities found in our study, can be obtained from our webpage [2].

Acknowledgments

We thank Lorenzo Alvisi (our shepherd) and the anonymous reviewers for their insightful comments. We thank members of ADSL, application developers and users, and file system developers, for valuable discussions. This material is based upon work supported by the NSF under CNS-1421033, CNS-1319405, and CNS-1218405 as well as donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Samsung, Sony, and VMware. Vijay Chidambaram and Samer Al-Kiswany are supported by the Microsoft Research PhD Fellowship and the NSERC Postdoctoral Fellowship, respectively. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] Necessary step(s) to synchronize filename operations on disk. http://austingroupbugs.net/view.php?id=672.

[2] Tool and Results: Application Crash Vulnerabilities. http://research.cs.wisc.edu/adsl/Software/alice/.

[3] Apache. Apache Zookeeper. http://zookeeper.apache.org/.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.8 edition, 2014.

[5] Steve Best. JFS Overview. http://jfs.sourceforge.net/project/pub/jfs.pdf, 2000.

[6] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[7] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. A history and evaluation of system r. *Communications of the ACM*, 24(10):632–646, 1981.

[8] Donald D Chamberlin, Arthur M Gilbert, and Robert A Yost. A history of system r and sql/data system. In *VLDB*, pages 456–464, 1981.

[9] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[10] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, February 2012.

[11] Chris Mason. Btrfs Mailing List. Re: Ordering of directory operations maintained across system crashes in Btrfs? http://www.spinics.net/lists/linux-btrfs/msg32215.html, 2014.

[12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[13] Jonathan Corbet. Solving the Ext3 Latency Problem. http://lwn.net/Articles/328363/, 2009.

[14] Jonathan Corbet. That massive filesystem thread. http://lwn.net/Articles/326471/, March 2009.

[15] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 329–342. ACM, 2013.

[16] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 307–320, Stevenson, Washington, October 2007.

[17] Bill Gallagher, Dean Jacobs, and Anno Langen. A High-performance, Transactional Filestore for Application Servers. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, pages 868–872, Baltimore, Maryland, June 2005.

[18] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[19] GNU. GNU Database Manager (GDBM). http://www.gnu.org.ua/software/gdbm/gdbm.html, 1979.

[20] Google. LevelDB. https://code.google.com/p/leveldb/, 2011.

[21] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[22] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[23] HyperSQL. HSQLDB. http://www.hsqldb.org/.

[24] Dean Kuo. Model and verification of a data manager based on aries. *ACM Trans. Database Syst.*, 21(4):427–479, December 1996.

[25] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, , and Jason Nieh. Pervasive Detection of Process Races in Deployed Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[26] Linus Torvalds. Git. http://git-scm.com/, 2005.

[27] Linus Torvalds. Git Mailing List. Re: what's the current wisdom on git over NFS/CIFS? http://marc.info/?l=git&m=124839484917965&w=2, 2009.

[28] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[29] Cris Pedregal Martin and Krithi Ramamritham. Toward formalizing recovery of (advanced) transactions. In *Advanced Transaction Models and Architectures*, pages 213–234. Springer, 1997.

[30] Chris Mason. The Btrfs Filesystem. oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf, September 2007.

[31] Matt Mackall. Mercurial. http://mercurial.selenic.com/, 2005.

[32] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[33] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[34] C Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[35] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[36] Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Coerced Cache Eviction and Discreet-Mode Journaling: Dealing with Misbehaving Disks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, China, June 2011.

[37] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[38] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[39] Frank Schmuck and Jim Wylie. Experience with transactions in quicksilver. In *ACM SIGOPS Operating Systems Review*, volume 25, pages 239–253. ACM, 1991.

[40] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[41] Stewart Smith. Eat My Data: How everybody gets file I/O wrong. In *OSCON*, Portland, Oregon, July 2008.

[42] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 29–42, San Francisco, CA, February 2009. USENIX Association.

[43] SQLite. SQLite transactional SQL database engine. http://www.sqlite.org/.

[44] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[45] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[46] Symas. Lightning Memory-Mapped Database (LMDB). `http://symas.com/mdb/`, 2011.

[47] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Joo-young Hwang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Towards Efficient, Portable Application-Level Consistency. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep '13)*, Farmington, PA, November 2013.

[48] The Open Group. POSIX.1-2008 IEEE Std 1003.1. `http://pubs.opengroup.org/onlinepubs/9699919799/`, 2013.

[49] The PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`.

[50] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[51] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[52] VMWare. VMWare Player. `http://www.vmware.com/products/player`.

[53] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[54] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

Torturing Databases for Fun and Profit

Mai Zheng[†] Joseph Tucek[‡] Dachuan Huang[†] Feng Qin [†] Mark Lillibridge[‡]
Elizabeth S. Yang[‡] Bill W. Zhao[‡] Shashank Singh[†]
[†] *The Ohio State University* [‡] *HP Labs*

Abstract

Programmers use databases when they want a high level of reliability. Specifically, they want the sophisticated ACID (atomicity, consistency, isolation, and durability) protection modern databases provide. However, the ACID properties are far from trivial to provide, particularly when high performance must be achieved. This leads to complex and error-prone code—even at a low defect rate of one bug per thousand lines, the millions of lines of code in a commercial OLTP database can harbor thousands of bugs.

Here we propose a method to expose and diagnose violations of the ACID properties. We focus on an ostensibly easy case: power faults. Our framework includes workloads to exercise the ACID guarantees, a record/replay subsystem to allow the controlled injection of simulated power faults, a ranking algorithm to prioritize where to fault based on our experience, and a multi-layer tracer to diagnose root causes. Using our framework, we study 8 widely-used databases, ranging from open-source key-value stores to high-end commercial OLTP servers. Surprisingly, all 8 databases exhibit erroneous behavior. For the open-source databases, we are able to diagnose the root causes using our tracer, and for the proprietary commercial databases we can reproducibly induce data loss.

1 Introduction

Storage system failures are extremely damaging—if your browser crashes you sigh, but when your family photos disappear you cry. Few people use process-pairs or n-versioning, but many use RAID.

Among storage systems, databases provide the strongest reliability guarantees. The atomicity, consistency, isolation, and durability (ACID) properties databases provide make it easy for application developers to create highly reliable applications. However, these properties come at a cost in complexity. Even the relatively simple SQLite database has more than 91 million lines of test code (including the repetition of parameterized tests with different parameters), which is over a thousand times the size of the core library itself [11]. Checking for the ACID properties under failure is notoriously hard since a failure scenario may not be conveniently reproducible.

In this paper, we propose a method to expose and diagnose ACID violations by databases under clean power faults. Unexpected loss of power is a particularly interesting fault, since it happens in daily life [31] and is a threat even for sophisticated data centers [24, 25, 27, 28, 29, 30, 41, 42] and well-prepared important events [18]. Further, unlike the crash model in previous studies (e.g., RIO [16] and EXPLODE [44]) where memory-page corruption can propagate to disk, and unlike the unclean power losses some poorly behaved devices suffer [46], a clean loss of power causes the termination of the I/O operation stream, which is the most idealized failure scenario and is expected to be tolerated by well-written storage software.

We develop four workloads for evaluating databases under this easy power fault model. Each workload has self-checking logic allowing the ACID properties to be checked for in a post-fault database image. Unlike random testing, our specifically designed workloads stress all four properties, and allow the easy identification of incorrect database states.

Given our workloads, we built a framework to efficiently test the behavior of databases under fault. Using a modified iSCSI driver we record a high-fidelity block I/O trace. Then, each time we want to simulate a fault during that run, we take the collected trace and apply the fault model to it, generating a new synthetic trace. We create a new disk image representative of what the disk state may be after a real power fault by replaying the synthetic trace against the original image. After restarting the database on the new image, we run the consistency checker for the

workload and database under test.

This record-and-replay feature allows us to systematically inject faults at every possible point during a workload. However, not all fault points are equally likely to produce failures. User-space applications including databases often have assumptions about what the OS, file system, and block device can do; violations of these assumptions typically induce incorrect behavior. By studying the errors observed in our early experiments, we identify five low-level I/O patterns that are especially vulnerable. Based on these patterns, we create a ranking algorithm that prioritizes the points where injecting power faults is most likely to cause errors; this prioritization can find violations about 20 times faster while achieving the same coverage.

Simply triggering errors is not enough. Given the huge code base and the complexity of databases, diagnosing the root cause of an error could be even more challenging. To help in diagnosing and fixing the discovered bugs, we collect detailed traces during the working and recording phases, including function calls, system calls, SCSI commands, accessed files, and accessed blocks. These multi-layer traces, which span everything from the block-level accesses to the workloads' high-level behavior, facilitate much better diagnosis of root causes.

Using our framework, we evaluate 8 common databases, ranging from simple open-source key-value stores such as Tokyo Cabinet and SQLite up to commercial OLTP databases. Because the file system could be a factor in the failure behavior, we test the databases on multiple file systems (ext3, XFS, and NTFS) as applicable. We test each combination with an extensive selection—exhaustively in some cases—of power-fault points. We can do this because our framework does not require the time-consuming process of running the entire workload for every fault, allowing us to emulate thousands of power faults per hour.

To our surprise, **all 8 databases exhibit erroneous behavior**, with 7 of the 8 clearly violating the ACID properties. In some cases, only a few records are corrupted, while in others the entire table is lost. Although advanced recovery techniques (often requiring intimate knowledge of the database and poorly documented options) may reduce the problem, not a single database we test can be trusted to keep all of the data it promises to store. By using the detailed multi-layer traces, we are able to pinpoint the root causes of the errors for those systems we have the source code to; we are confident that the architects of the commercial systems could quickly correct their defects given similar information.

In summary, our contributions are:

- **Carefully designed workloads and checkers to test the ACID properties of databases.** Despite extensive test suites, existing databases still contain bugs. It is a non-trivial task to verify if a database run is correct after fault injection. Our 4 workloads are carefully designed to stress different aspects of a database, and further are designed for easy validation of correctness.

- **A cross-platform method for exposing reliability issues in storage systems under power fault.** By intercepting in the iSCSI layer, our framework can test databases on different operating systems. By recording SCSI commands (which are what disks actually see), we can inject faults with high fidelity. Further, SCSI tracing allows systemic fault injection and ease of repeating any error that is found. Although we focus here on databases, this method is applicable to any software running on top of a block device.

- **A pattern-based ranking algorithm to identify the points most likely to cause problems when power faulted in database operations.** We identify 5 low-level patterns that indicate the most vulnerable points in database operations from a power-fault perspective. Further analysis of the root causes verifies that these patterns are closely related to incorrect assumptions on the part of database implementers. Using these patterns to prioritize, we can accelerate testing by 20 times compared to exhaustive testing while achieving nearly the same coverage.

- **A multi-layer tracing system for diagnosing root causes of ACID violations under fault.** We combine the high-level semantics of databases with the low-level I/O traffic by tracing the function calls, system calls, SCSI commands, files, and I/O blocks. This correlated multi-layer trace allows quick diagnosis of complicated root causes.

- **Experimental results against 8 popular databases.** We apply our framework to a wide range of databases, including 4 open-source and 4 commercial systems. All 8 databases exhibit some sort of erroneous behavior. With the help of the multi-layer tracer, we are able to quickly pinpoint the root causes for the open-source ones.

2 Design Overview

2.1 Fault Model

We study the reliability of databases under a simple fault model: clean power fault. Specifically, we model loss of power as the potential loss of any block-level operations

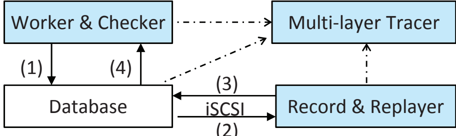


Figure 1: Overall Workflow

that have not been acknowledged as committed to persistent storage, with the proviso that if we drop an operation, we must drop any operations that explicitly depend on it as well. Although our system can induce the more complex faults of other fault models [16, 44, 46], which consider more complex data corruption and inconsistent behavior, we focus here on the simplest case. Such corruption and reordering/dropping of operations seem unreasonable to expect databases to tolerate, given that they so badly violate the API contract of the lower-level storage system.

The specifics of our fault model are as follows: a fault can occur at any point, the size of data written to media is always an integer block multiple (512 bytes or 4 KB), the device keeps all promises of durable commit of operations (e.g., completed `syncs` are honored as are write barriers), blocks are written (or not) without corruption, and interrupted or dropped operations have no effect. We believe this is the most idealized scenario under power failure, and we expect databases to honor their ACID guarantees under such faults.

2.2 Overall Workflow

Our testing framework injects simulated power faults by intercepting iSCSI [4] commands. iSCSI is a standard allowing one machine (the initiator) to access the block storage of another machine (the target) through the network. To everything above the block driver on the initiator, iSCSI is completely transparent, making it an ideal place to intercept disk activity. Further, the iSCSI target daemon can be implemented in user space [6], which greatly increases the flexibility of fault injection compared to a more complex kernel-level failure emulation layer. Finally, iSCSI interception allows a single fault injection implementation to test databases running on any operating system having an iSCSI initiator (aka all modern OSes).

Figure 1 shows an overview of our framework. There are three main components: the worker and checker, a record and replayer, and a multi-layer tracer. The overall workflow goes as follows: First, the worker applies a workload to stress a database starting from a known disk image. Rather than simply randomly writing to the database, the workloads are carefully designed so that the checker can verify the ACID properties of the post-fault image. Second, the record and replayer monitors the disk activities via the iSCSI layer. All blocks in the data transfer commands are recorded. Third, with the recorded block trace, the replayer simulates a power fault by partially replaying the block operations based on fault injection policies against a copy of the starting image, creating a post-fault disk image. Fourth, the checker opens the database on the post-fault image and verifies the ACID properties of the recovered database. During each of the above steps, the multi-layer tracer traces database function calls, system calls, SCSI commands, accessed files, and accessed blocks to provide unified information to diagnose the root cause of any ACID violations.

3 Worker and Checker

We developed four workloads with different complexities to check if a database provides ACID properties even when under fault. In particular, we check (1) atomicity, which means a transaction is committed "all or nothing"; (2) consistency, which means the database and application invariants always hold between transactions; (3) isolation, which means the intermediate states of a transaction are not visible outside of that transaction; and (4) durability, which means a committed transaction remains so, even after a power failure or system crash.

Each workload stresses one or more aspects of the databases including large transactions, concurrency handling, and multi-row consistency. Each workload has self-checking properties (e.g., known values and orders for writes) that its associated checker uses to check the ACID properties. For simplicity, we present pseudo code for a key-value store; the equivalent SQL code is straightforward. We further log timestamps of critical operations to a separate store to aid in checking.

Workload 1: A single thread performs one transaction that creates `txn_size` (a tunable parameter) rows:

```
Begin Transaction
  for i = 1 to txn_size do
    key   = "k-" + str(i)
    value = "v-" + str(i)
    put(key, value)
  end
  before_commit = get_timestamp()
Commit Transaction
after_commit    = get_timestamp()
```

We use this workload to see whether large transactions (e.g., larger than one block) trigger errors. The two timestamps `before_commit` and `after_commit` record the time boundaries of the commit. The checker for this workload is straightforward: if the fault was after the commit (i.e., later than `after_commit`), check that all

`txn_size` rows are present; otherwise, check that none of the rows are present. If only some rows are present, this is an atomicity violation. Being a single-threaded workload, isolation is not a concern. For consistency checking, we validate that a range scan gives the same result as explicitly requesting each key individually in point queries. Moreover, each retrievable key-value pair should be matching; that is, key N should always be associated with value N.

We report a durability error if the fault was clearly after the commit and the rows are absent. A corner case here is if the fault time lies between `before_commit` and `after_commit`. The ACID guarantees are silent as to what durability properties an in-flight commit has (the other properties are clear) so we do not report durability errors in such cases. We do find that some databases may change back and forth between having and not having a transaction as the point we fault advances during a commit.

Workload 2: This is a multi-threaded version of workload 1, with each thread including its thread ID in the key. Since we do not have concurrent transactions operating on overlapping rows, we do not expect isolation errors. We do, however, expect the concurrency handling of the database to be stressed. For example, one thread may call `msync` and force state from another thread to disk unexpectedly.

Workload 3: This workload tests single-threaded multi-row consistency by simulating concurrent, non-overlapping banking transactions. It performs `txn_num` transactions sequentially, each of which moves money among a different set of `txn_size` accounts. Each account starts with some money and half of the money in each even numbered account is moved to the next higher numbered account (i.e., half of k-t-$2i$'s money is moved to k-t-$(2i+1)$ where t is the transaction ID):

```
for t = 1 to txn_num do
  key_prefix = "k-" + str(t) + "-"
  Begin Transaction
    for i in 1 to txn_size/2 do
      k1 = key_prefix + str(2*i)
      k2 = key_prefix + str(2*i+1)
      tmp1 = get(k1)
      put(k1, tmp1 - tmp1/2)
      tmp2 = get(k2)
      put(k2, tmp2 + tmp1/2)
    end
    before_commit[t] = get_timestamp()
  Commit Transaction
  after_commit[t]    = get_timestamp()
end
```

As with workload 1, we check that each transaction is all-or-nothing, that transactions committed before the fault are present (and those after are not), and that the results for range-scans and point-queries match. Further, since none of the transactions change the total amount of money, the checker tests every pair of rows with keys of the form k-t-$2i$, k-t-$(2i+1)$ to see whether their amounts sum to the same value as in the initial state.

	key	value
meta rows	THR-1-TXN-1	k-2-k-5-TS-00:01
	THR-1-TXN-2	k-6-k-8-TS-00:13
	THR-2-TXN-1	k-7-k-6-TS-00:03
	THR-2-TXN-2	k-3-k-7-TS-00:14
work rows	k-1	v-init-1
	k-2	v-THR-1-TXN-1
	k-3	v-THR-2-TXN-2
	k-4	v-init-4
	k-5	v-THR-1-TXN-1
	k-6	v-THR-1-TXN-2
	k-7	v-THR-2-TXN-2
	k-8	v-THR-1-TXN-2

Figure 2: An example workload 4 output table.

Workload 4: This is the most stressful (and time-consuming) workload. It has multiple threads, each performing multiple transactions, and each transaction writes to multiple keys. Moreover, transactions from the same or different threads may update the same key, fully exercising database concurrency control.

Figure 2 shows an example output table generated by workload 4. In this example, there are two threads (`THR-1` and `THR-2`), each thread contains two transactions (`TXN-1` and `TXN-2`), and each transaction updates two work-row keys (e.g., `k-2` and `k-5` for `THR-1-TXN-1`). The table has two parts: the first 4 rows (meta rows) keep the metadata about each transaction, including the non-meta keys written in that transaction and a timestamp taken immediately before the commit. The next 8 rows (work rows) are the working region shared by the transactions. Each transaction randomly selects two keys within the working region, and updates their values with its thread ID and transaction ID. For example, the value `v-THR-1-TXN-1` of the key `k-2` means the transaction 1 in thread 1 updated the key `k-2`. All rows start with initial values (e.g., `v-init-1`) to give a known starting state.

The following pseudo-code shows the working logic of the first transaction of thread one; the runtime values in the comments are the ones used to generate Figure 2:

```
me = "THR-1-TXN-1"
Begin Transaction
  // update two work rows:
  key1 = get_random_key()          //k-2
```

```
  put(key1, "v-" + me)
  key2 = get_random_key()        //k-5
  put(key2, "v-" + me)

  // update my meta row:
  before_commit = get_timestamp() //TS-00:01
  v = key1 + "-" + key2 + "-"
    + str(before_commit)  //k-2-k-5-TS-00:01
  put(me, v)
Commit Transaction
after_commit[me] = get_timestamp()
```

Each transaction involves multiple rows (one meta row and multiple work rows) and stores a large amount of self-description (i.e., the meta row records the work rows updated and the work rows specify which meta row is associated with their last updates). The `after_commit` timestamps allow greater precision in identifying durability, but are not strictly necessary.

As with workloads 1–3, we validate that range and point queries match. In addition, since the table contains initial values (e.g., `k-1 = v-init-1`) before the workload starts, we expect the table should at least maintain the original values in the absence of updates—if any of the initial rows are missing, we report a durability error. A further quick check verifies that the format of each work row is either in the initial state (e.g., `k-1 = v-init-1`), or was written by a valid transaction (e.g., `k-2 = v-THR-1-TXN-1`). Any violation of the formatting rules is a consistency error.

Another check involves multiple rows within each transaction. Specifically, the work rows and meta rows we observe need to match. When we observe at least one row (either a work or a meta row) updated by a transaction T_a and there is a missing row update from T_a, we classify the potential errors based on the missing row update: If that row is corrupted (unreadable), then we report a durability error. If furthermore the transaction T_a definitely committed after the fault point (i.e., T_a's `before_commit` is after the fault injection time), we also report an isolation error because the transaction's uncommitted data became visible.

Alternatively, if the row missing the update (from transaction T_a) contains either the initial value or the value from a transaction T_b known to occur earlier (i.e., transaction T_b's `after_commit` is before transaction T_a's `before_commit`), then we report an atomicity error since partial updates from transaction T_a are observed. If furthermore transaction T_a definitely committed before the fault point, we also report a durability error, and if it definitely committed after the fault point we report an isolation error.

Note that because each transaction saves timestamps, we can determine if a work row might have been legitimately overwritten by another transaction. As shown in Figure 2, the first transaction of thread 2 (`THR-2-TXN-1`) writes to `k-7` and `k-6`, but the two rows are overwritten by `THR-2-TXN-2` and `THR-1-TXN-2`, respectively. Based on the timestamp bounds of the commits, we can determine if these overwritten records are legitimate. One last check is for transactions that definitely committed but do not leave any update; we report this as a durability error.

4 Record and Replay

The record-and-replay component records the I/O traffic generated by the Worker under the workload, and replays the I/O trace with injected power faults. As mentioned in Section 2.2, the component is built on the iSCSI layer. This design choice gives fine-grained and high-fidelity control over the I/O blocks, and allows us to transparently test databases across different OSes.

4.1 Record

Figure 3(a) shows the workflow for the record phase. The Worker exercises the database, which generates filesystem operations, which in turn generate iSCSI requests that reach the backing store. By monitoring the iSCSI target daemon, we collect detailed block I/O operations at the SCSI command level. More specifically, for every SCSI command issued to the backing store, the SCSI Parser examines its command descriptor block (CDB) and determines the command type. If the command causes data transfer from the initiator to the target device (e.g., `WRITE` and its variants), the parser records the timestamp of the command and further extracts the logical block address (LBA) and the transfer length from the CDB, then invokes the Block Tracer. The Block Tracer fetches the blocks to be transferred from the daemon's buffer and records it in an internal data log. The command timestamp, the LBA, the transfer length, and the offset of the blocks within the data log are further recorded in an index log for easy retrieval. In this way, we obtain a sequence of block updates (i.e., the Worker's block trace) that can be used to generate a disk image representative of the state after a power fault.

The block trace collected from SCSI commands is enough to generate simulated power faults. However, given the huge number of low-level block accesses in a trace, how to inject power faults efficiently is challenging. Moreover, the block I/Os themselves are too low-level to infer the high-level operations of the database under testing, which are essential for understanding why an ACID violation happens and how to fix it. To address these challenges, we design a multi-layer tracer, which correlates the low-level block accesses with various high-level semantics.

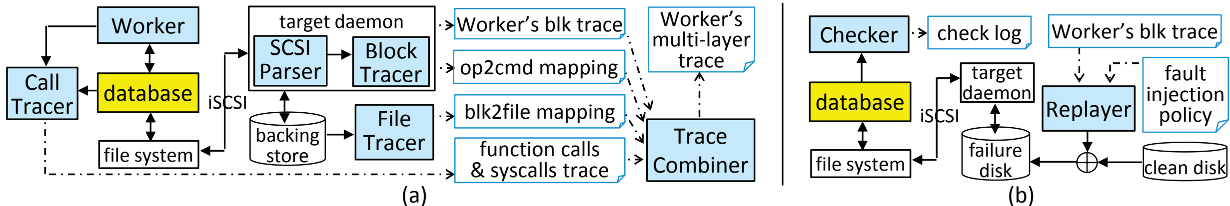


Figure 3: (a) Record phase (b) Replay for testing

In particular, we collect three more types of traces in the record phase. First, our op2cmd mapper (inside the Block Tracer, not shown) maps each block operation (512 B or 4 KB) to the specific SCSI command it is part of (a single SCSI command can cover over a megabyte); the resulting mapping (i.e., op2cmd mapping) lets us infer which operations the file system (or database) treated as one logical unit and thus may be correlated.

Second, the File Tracer connects the blocks being updated to the higher-level files and directories they store. For a given file system, the meaning of each block is well-defined (e.g., for ext3 the location of the inode bitmap is fixed after formatting, and the blocks belonging to a particular file are defined via the corresponding inode block). The File Tracer first identifies the inode number for each file (including regular files, directories, and the filesystem journal). Then, it extracts the corresponding data block numbers for each inode. In this way, the File Tracer generates a mapping (the blk2file mapping) that identifies the ownership of each block in the file system. Because each database file usually has a well-defined functionality (e.g., UNDO/REDO log files), the blk2file mapping lets us identify which I/O requests are modifying, say, the UNDO log versus the main index file. We can also identify which updates are to filesystem metadata and which are to the filesystem journal. This trace, together with the op2cmd mapping above, gives us an excellent picture of the behavior of the database at the I/O level.

The third type of trace is generated by the Call Tracer, which instruments the workload and the database and records all function calls and system calls invoked. Each call is recorded with an invoke timestamp, a return timestamp, and its thread ID. This information not only directly reveals the semantics of the databases, but also helps in understanding the lower-level I/O traffic. For example, it allows us to tell if a particular I/O was explicitly requested by the database (e.g., by correlating `fsync` and `msync` calls with their respective files) or if it was initiated by the file system (e.g., dirty blocks from another file or ordinary dirty block write back).

Finally, all of the traces, including the block trace, op2cmd mapping, blk2file mapping, and the call trace, are supplied to the Trace Combiner. The block trace and call trace are combined based on the timestamps associated with each entry. For example, based on the timestamps when a `fsync` call started and finished, and the timestamp when a SCSI WRITE command is received in between, we associate the blocks transferred in the WRITE command with the `fsync` call. Note that in a multi-threaded environment, the calls from different threads (which can be identified by the associated thread IDs) are usually interleaved. However, for each synchronous I/O request (e.g., `fsync`), the blocks transferred are normally grouped together without interference from other requests via a write barrier. So in practice we can always associate the blocks with the corresponding synchronous calls. Besides combining the block trace and the call trace, the op2cmd mapping and the blk2file mapping are further combined into the final trace based on the LBA of the blocks. In this way, we generate a multi-layer trace that spans everything from the highest-level semantics to the lowest-level block accesses, which greatly facilitates analysis and diagnosis. We show examples of the multi-layer traces in Section 5.1.

4.2 Replay for Testing

After the record phase, the replayer leverages the iSCSI layer to replay the collected I/O block trace with injected faults, tests whether the database can preserve the ACID-properties, and helps to further diagnose the root causes if a violation is found.

4.2.1 Block Replayer

Figure 3(b) shows the workflow of the replay-for-testing phase. Although our replayer can inject worse errors (e.g., corruption, flying writes, illegally dropped writes), we focus on a "clean loss of power" fault model. Under this fault model, all data blocks transferred before the power cut are successfully committed to the media, while others are lost. The Replayer first chooses a fault point based on the injection policy (see Section 4.2.2). By starting with a RAM disk image of the block device at the beginning of the workload, we produce a post-fault image by selectively replaying the operations recorded in the Worker's block trace. This post-fault image can then

op#	LBA	file
...	...	...
35	*1038*	*x.db*
36	2347	x.db
37	2351	x.db
...	...	...
49	1038	x.db

(a) P_{rep}

op#	LBA	file
61	3142	x.db
62	3146	x.db
63	*2081*	*x.db*
64	*5191*	*x.db*
65	1025	x.db
66	1029	x.db

(b) P_{jump}

op#	LBA	cmd#
152	1070	42
153	*1106*	*43*
154	*1110*	*43*
155	*1114*	*43*
156	1118	43
157	1765	44

(c) P_{head}

op#	LBA	file
245	5545	x.db
246	5646	x.db
247	*5545*	*x.db*
248	*8351*	*fs-j*
249	*8352*	*fs-j*
250	8356	fs-j

(d) P_{tran}

op#	LBA	file	syscall
331	1012	x.db	msync(x.db)
...	...	...	...
460	6841	x.log	fsync(x.log)
461	9598	fs-j	fsync(x.log)
462	9602	fs-j	fsync(x.log)
463	*1012*	*x.db*	*fsync(x.log)*

(e) P_{mmap}

Figure 4: Five patterns of vulnerable points based on real traces: (a) repetitive LBA, (b) jumping LBA, (c) head of command, (d) transition between files, and (e) unintended update to `mmap`'ed block. op# means the sequence number of the transfer operation for a data block, cmd# means the SCSI command's sequence number, `x.db` is a blinded database file name, `x.log` is a blinded database log file name, and `fs-j` means filesystem journal. The red italic lines mean a power fault injected immediately after that operation may result in failures (a–d), or a fault injected after later operations may result in failures (e). For simplicity, only relevant tracing fields are shown.

be mounted for the Checker to search for ACID violations. Because our power faults are simulated, we can reliably replay the same block trace and fault deterministically as many times as needed.

4.2.2 Fault Injection Policy

For effective testing, we design two fault injection policies that specify where to inject power faults given a workload's block trace.

Policy 1: Exhaustive. Under our fault model, each block transfer operation is atomic, although multi-block SCSI operations are not. The exhaustive policy injects a fault after every block transfer operation, systematically testing all possible power-fault scenarios for a workload.

Although exhaustive testing is thorough, for complicated workloads it may take days to complete. Randomly sampling the fault injection points may help to reduce the testing time, but also reduce the fault coverage. Hence we propose a more principled policy to select the most vulnerable points to inject faults.

Policy 2: Pattern-based ranking. By studying the multi-layer traces of two databases (TokyoCabinet and MariaDB) with exhaustive fault injection, we extracted five vulnerable patterns where a power fault occurring likely leads to ACID violations. In particular, we first identify the correlation between the fault injection points and the ACID violations observed. Then, for each fault point leading to a violation, we analyze the context information recorded in the trace around the fault point and summarize the patterns of vulnerable injection points.

Figure 4 shows examples of the five patterns based on the real violations observed in our early experiments:

Pattern A: repetitive LBA (P_{rep}). For example, in Figure 4(a) op#35 and op#49 both write to LBA 1038, which implies that 1038 may be a fixed location of important metadata. The parameter for this pattern is the repetition threshold.

Pattern B: jumping LBA sequence (P_{jump}). In Figure 4(b), the operations before op#63 access a large contiguous region (e.g., op#62 , op#61, and earlier operations which are not shown), and the operations after op#64 are also contiguous. The LBAs of op#63 and op#64 are far away from that of the neighbor operations and are jumping forward (e.g., from 2081 to 5191) or backward (e.g., from 5191 to 1025). This may imply switching operation or complex data structure updates (e.g., after appending new nodes, update the metadata of a B+ tree stored at the head of a file). The parameters of this pattern include jumping distance and jumping direction.

Pattern C: head of a SCSI command (P_{head}). Each SCSI command may transfer multiple blocks. For example, in Figure 4(c), op#153–156 all belong to cmd#43. If the fault is injected after op#153, 154, or 155, the error will be triggered. The reason may be that the blocks transferred in that SCSI command need to be written atomically, which is blocked by these fault points. The parameter of this pattern is the minimal length of the head command.

Pattern D: transition between files (P_{tran}). In Figure 4(d), the transition is between a database file (`x.db`) and the filesystem journal (`fs-j`). This pattern may imply an interaction between database and file system (e.g., delete a log file after commit) that requires special coding. The pattern also includes transitions among database files because each database file usually has a specific function (e.g., UNDO/REDO logs) and the transition may imply some complex operations involving multiple files.

Pattern E: unintended update to `mmap`'ed blocks (P_{mmap}). `mmap` maps the I/O buffers for a portion of a file

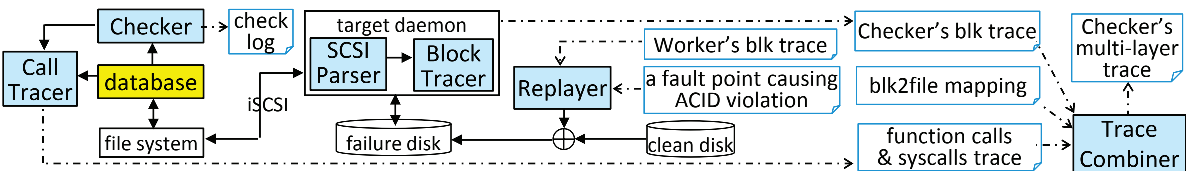


Figure 5: Replay for diagnosis

into the application's address space and thus allows the application to write to the file cache directly. This mechanism is attractive to database systems because it allows the database to circumvent double-buffering of the file cache when running on top of a file system [22]. As shown in Figure 4(e), `x.db` is a `mmap`'ed file, and LBA 1012 is in the `mmap`'ed region. Based on the system call trace, we find that op#331 is caused by calling `msync` on the memory map of `x.db`, which is an explicit request of the database. On the other hand, op#463, which also updates LBA 1012, is unintended because it happens when calling `fsync` on another file (`x.log`). Other causes for such implicit updates could be the periodic write back of dirty pages or dirty page write back due to memory pressure. In the real trace, injecting a fault immediately after the implicit update (i.e., op#463 in this case) may not necessarily cause ACID violations. Instead, injecting faults later may cause failures. So the pattern considers all operations between the implicit update and the next explicit update, and uses sampling to select fault injection points within the range. The parameter of this pattern is the sampling rate.

In summary, we extract the five block-level patterns from the real failure logs of two databases. Three of them (P_{rep}, P_{jump}, and P_{head}) are independent of file systems and OSes, two (P_{tran} and P_{mmap}) require knowledge of the filesystem structure, and P_{mmap} is further associated with system calls. Intuitively, these patterns capture the critical points in the I/O traffic (e.g., updates to some fixed location of metadata or transition between major functions) so we use them to guide fault injection. Specifically, after obtaining the traces from the record phase, we check them against the patterns. For each fault injection point, we see if it matches any of the five patterns, and score each injection point based on how many of the patterns it matches. Because a fault is always injected after a block is transferred, we use the corresponding transfer operation to name the fault injection point. For example, in Figure 4(a) op#35 and op#49 match the repetitive pattern (assuming the repetition threshold is 2), so each of them has the score of 1, while op#36 and op#37 remain 0. An operation can gain a score of more than 1 if it matches multiple patterns. For example, an operation may be simultaneously a repetitive operation (P_{rep}), the first operation of a command (P_{head}), and represent a transition between files (P_{tran}), yielding a score of 3. After scoring the operations, the framework injects power faults at the operations with the highest score. By skipping injection points with low scores, pattern-based ranking reduces the number of fault injection rounds needed to uncover failures. We show real examples of the patterns and the effectiveness and efficiency of this fault injection policy in Section 5.

4.3 Replay for Diagnosis

Although identifying that a database has a bug is useful, diagnosing the underlying root cause is necessary in order to fix it. Figure 5 shows the workflow of our diagnosis phase. Similar to the steps in replay for testing, the replayer replays the Worker's block trace up to the fault point (identified in the replay for testing phase) that lead to the ACID failure. Again, the Checker connects to the database and verifies the database's state. However, for diagnosis we activate the full multi-layer tracing. Moreover, the blocks read by the database in this phase are also collected because they are closely related to the recovery activity. The Checker's block trace, blocks-to-files mapping (collected during the record phase), and the function calls and system calls trace are further combined into Checker's multi-layer trace. Together with the check log of the integrity checker itself, these make identifying the root cause of the failure much easier. Further exploring the behavior of the system for close-by fault points that do not lead to failure [45] can also help. We discuss diagnosis based on the multi-layer traces in more detail in Section 5.1.

5 Evaluation

We built our prototype based on the Linux SCSI target framework [6]. The Call Tracer is implemented using PIN [8]. The File Tracer is built on `e2fsprogs` [3] and XFS utilities [12]. We use RHEL 6[1] as both the iSCSI target and initiator to run the databases, except that we use Windows 7 Enterprise to run the databases using NTFS.

We apply the prototype to eight widely-used databases, including three open-source embedded

[1] All results were verified with Debian 6, and per time constraints a subset were verified with Ubuntu 12.04 LTS.

databases (TokyoCabinet, LightningDB, and SQLite), one open-source OLTP SQL server (MariaDB), one proprietary licensed[2] key-value store (KVS-A), and three proprietary licensed OLTP SQL databases (SQL-A, SQL-B, and SQL-C). All run on Linux except SQL-C, which runs on NTFS.

Since none of the tested databases fully support raw block device access,[3] the file system could be another factor in the failure behavior. Hence for our Linux experiments, we tested the databases on two different file systems: the well-understood and commonly deployed ext3, and the more robust XFS. We have not yet fully implemented the Call Tracer and the File Tracer for Windows systems but there are no core technical obstacles in implementing these components using Windows-based tooling [7, 9]. Also, for the proprietary databases without debugging symbols, we supply limited support for diagnosis (but full support for testing).

5.1 Case Studies

In this subsection, we discuss three real ACID-violation cases found in three databases, and show how the multi-layer traces helped us quickly diagnose their root causes.

5.1.1 TokyoCabinet

When testing TokyoCabinet under Workload 4, the Checker detects the violations of atomicity (a transaction is partially committed), durability (some rows are irretrievable), and consistency (the retrievable rows by the two query methods are different). These violations are non-deterministic—they may or may not manifest themselves under the same workload—and the failure symptoms vary depending on the fault injection point, making diagnosis challenging. The patterns applicable to this case include P_{jump}, P_{head}, P_{tran}, and P_{mmap}.

For a failing run, we collect the Checker's multi-layer trace (Figure 6(b)). For comparison purposes, we also collect the Checker's trace for a bug-free run (Figure 6(a)). By comparing the two traces, we can easily see that `tchdbwalrestore` is not invoked in the failing run. In the parent function (`tchdbopenimpl`) there is a read of 256 bytes from the database file `x.tcb` in both traces, but the content read is different by one bit (i.e., 108 vs. 118 in op#1). Further study of the data structures defined in the source code reveals that the first 256 bytes contain a flag (`hdb->flags`), which determines whether to

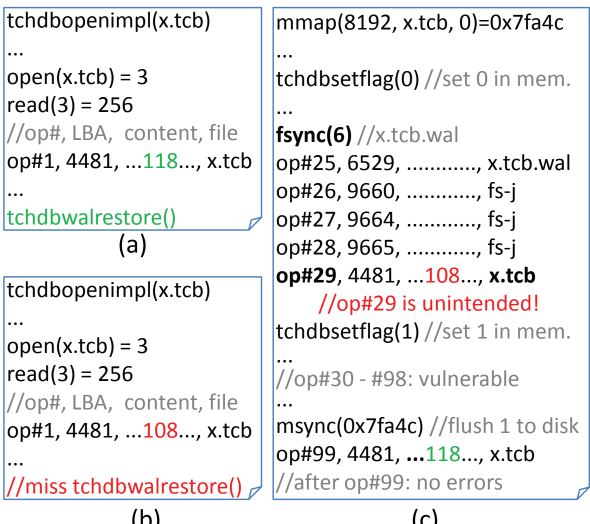


Figure 6: Example of multi-layer traces adapted from the real traces of TokyoCabinet. (a) Checker's trace when no violations were found. (b) Checker's trace when ACID violations were found. (c) Worker's trace around the power fault points leading to ACID violations. LBA, and address) are reduced and only relevant fields and lines are shown.

invoke `tchdbwalrestore` on startup. The one bit difference in op#1 implies that some write to the beginning of `x.tcb` during the workload causes this ACID violation.

We then look at the Worker's multi-layer trace near the power-fault injection points that manifest this failure (Figure 6(c)). The majority of the faults within op#30–98 cause ACID violations, while power losses after op#99 do not cause any trouble. So the first clue is op#99 changes the behavior. Examining the trace, we notice that the beginning of `x.tcb` is `mmap`'ed, and that op#99 is caused by an explicit `msync` on `x.tcb` and sets the content to 118. By further examining the writes to `x.tcb` before op#30–98, we find that op#29 also updates `x.tcb` by setting the content to 108. However, this block update is unintended: an `fsync` on the write-ahead log `x.tcb.wal` triggers the OS to also write out the dirty block of `x.tcb`.

The whole picture becomes more clear with the collected trace of high-level function calls. It turns out that at the beginning of each transaction (`tchdbtranbegin()`, not shown), the flag (`hdb->flags`) is set to 0 (`tchdbsetflag(0)`), and then set to 1 (`tchdbsetflag(1)`) after syncing the initial `x.tcb.wal` to disk (`fsync(6)`). If the synchronization of `x.tcb.wal` with disk is successful, the flag 0 should be invisible to disk. In rare cases, however, the `fsync` on `x.tcb.wal` causes an unintended flush of the flag 0 to `x.tcb` on disk (as captured by op#29). In this scenario, if

[2] Due to the litigious nature of many database vendors (see "The DeWitt Clause" [1]), we are unable to identify the commercial databases by name. We assure readers that we tested well recognized, well regarded, and mainstream software.

[3] Nearly all modern databases run through the file system. Of the major commercial OLTP vendors, Oracle has removed support for raw storage devices [33], IBM has deprecated it for DB2 [23], and Microsoft strongly discourages raw partitioning for SQL Server [26].

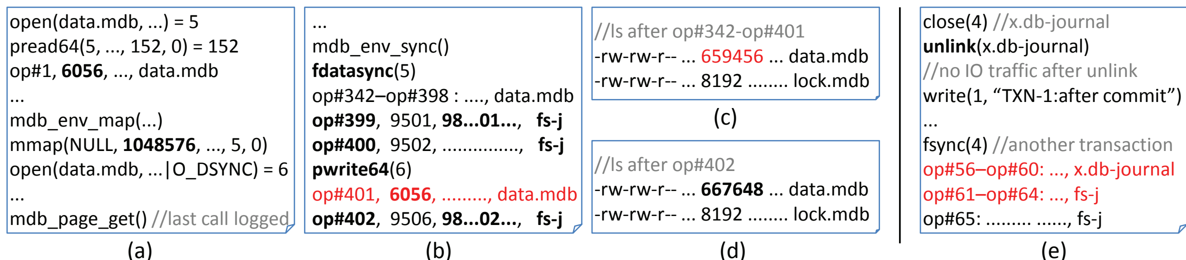


Figure 7: Examples of multi-layer traces adapted from real traces of LightningDB (a–d) and SQLite (e): (a) LightningDB Checker's trace, (b) Worker's trace around the bug-triggering fault point (op#401), (c) Checker's check log showing the size of `data.mdb` after op#342–401, (d) Checker's check log showing the size of `data.mdb` after op#402, and (e) SQLite Worker's trace around the fault points (op#56–64) that cause durability violations.

a power fault is injected at the points between op#29 and 99, it will interrupt the transaction and the post-fault disk image has the flag set to 0. Upon the recovery from the post-fault disk image, the database will mistakenly think `x.tcb.wal` has not been initialized (because the on-disk flag is 0), and skip the `tchdbwalrestore()` procedure. Consequently, an incomplete transaction is exposed to the user.

The unintended updates to the `mmap`'ed file could be caused for two reasons. One is the flushing of dirty pages by kernel threads due to memory pressure or timer expiry (e.g., `dirty_writeback_centisecs`), the other is the internal dirty page flushing routines of file systems. Since we can see the `fsync` call that causes the unintended update, it is clear that the fact that ext3 and XFS are aggressive in flushing blocks when there is a sync is to blame. However, regardless if it is due to the kernel or the file system, the programmer's incorrect assumption that a change to a `mmap`'ed files will not be asynchronously written back is the underlying root cause.

One solution to this problem is using failure-atomic `msync` [34], which prevents the non-explicit write back to the `mmap`'ed file.

5.1.2 LightningDB

When testing LightningDB on ext3 under Workload 4, the Checker, which links the database library into its address space, crashes on certain queries. The applicable patterns include P_{rep}, P_{head}, and P_{tran}.

The Checker's multi-layer trace (Figure 7(a)) shows that before the crash there are two `pread64`s (the 2nd one is omitted in the figure) from the head (LBA 6056) of the database file `data.mdb`; also, `data.mdb` is `mmap`'ed into memory. The size of the mapping is 1,048,576 bytes (256 4 KB-pages), which exceeds the actual length of the file. The last function logged before the crash is `mdb_page_get`, which returns the address of a page. The LightningDB documents [5] and source code reveal that the first two pages of this file are metapages, which maintain the valid page information of the internal B+ tree, and that the mapping is 256 pages by default for performance reasons. Given this information, we suspect that the crash is caused by referencing a `mmap`'ed page that is valid based on the metapages but lies beyond the end of the backing file.

The Worker's trace (Figure 7(b)) and the Checker's check logs ((c) and (d)) verify our hypothesis. In this example, a power fault after op#401 leads to the crash, while a fault any other time (e.g., after op#342–400 and op#402) causes no problem. As shown in (b), op#401 is an update to the head (LBA 6056) of `data.mdb`, which maintains the valid page information. However, after applying op#401, the size of `data.mdb` did not get updated in the file system (Figure 7 (c)). Only after op#402, which is an update to the filesystem metadata, is the length of the file increased to 667,648 bytes (Figure 7 (d)), and the memory-mapping is safe to access from then on.

Based on the traces we can further infer that op#399, 400, and 402 form a filesystem journal transaction that updates the length metadata for `data.mdb`. The content of op#399 (i.e., "98...01", which is 98393bc001 in the real trace) matches the magic number of the ext3 journal and tells us that it is the first block (i.e., the descriptor block) of the journal transaction. op#402 (with content "98...02") is the last block (i.e., the commit block) of the journal transaction. op#400 is a journaled length update that matches the format of the inode, the superblock, the group descriptor, and the data block bitmap, all of which need to be updated when new blocks are allocated to lengthen a file. This is why the length update is invisible in the file system until after op#402: without the commit block, the journal transaction is incomplete and will never be checkpointed into the main file system.

Note that op#401 itself does not increase the file size since it is written to the head of the file. Instead, the file is extended by op#342–398, which are caused by appending B+ tree pages in the memory (via `lseeks` and

pwrites before the mdb_env_sync call) and then calling fdatasync on data.mdb. On ext3 with the default "ordered" journaling mode, the file data is forced directly out to the main file system prior to its metadata being committed to the journal. This is why we observe the journaling of the length update (op#399, 400, and 402) after the file data updates(op#342–398).

The fact that the journal commit block (op#402) is flushed with the next pwrite64 in the same thread means fdatasync on ext3 does not wait for the completion of journaling (similar behavior has been observed on ext4). LightningDB's aggressive mmap'ing and referencing pages without verifying the backing file finally trigger the bug. We have verified that changing fdatasync to fsync fixes the problem. Another platform-independent solution is checking the consistency between the metapage and the actual size of the file before accessing.

Because we wanted to measure how effective our system is in an unbiased manner, we waited to look at LightningDB until after we had finalized and integrated all of our components. It took 3 hours to learn the LightningDB APIs and port the workloads to it, and once we had the results of testing, it took another 3 hours to diagnose and understand the root cause. As discussed in more detail in Section 5.3, it takes a bit under 8 hours to do exhaustive testing for LightningDB, and less than 21 minutes for pattern-based testing. Given that we had no experience with LightningDB before starting, we feel this shows that our system is very effective in finding and diagnosing bugs in databases.

5.1.3 SQLite

When testing SQLite under any of the four workloads, the Checker finds that a transaction committed before a power fault is lost in the recovered database. The applicable patterns include P_{rep}, P_{head}, and P_{tran}.

Figure 7(e) shows the Worker's multi-layer trace. At the end of a transaction (TXN-1), the database closes and unlinks the log file (x.db-journal), and returns to the user immediately as implied by the "after commit" message. However, the unlink system call only modifies the in-memory data structures of the file system. This behavior is correctly captured in the trace—i.e., no I/O traffic was recorded after unlink. The in-memory modification caused by unlink remains volatile until after completing the first fsync system call in the next transaction, which flushes all in-memory updates to the disk. The SQLite documents [10] indicate that SQLite uses an UNDO log by default. As a result, when a power fault occurs after returning from a transaction but before completing the next fsync (i.e., before op#65), the UNDO log of the transaction remains visible on the disk (instead of being unlinked). When restarting the database, the committed transaction is rolled back unnecessarily, which makes the transaction non-durable.

One solution for this case is to insert an additional fsync system call immediately after the unlink, and do not return to the user until the fsync completes. Note that this case was manifested when we ran our experiments under the default DELETE mode of SQLite. The potential non-durable behavior is, surprisingly, known to the developers [2]. We ran a few additional experiments under the WAL mode as suggested by the developers, and found an unknown error leading to atomicity violation. We do not include the atomicity violation in Table 1 (Section 5.2) since we can reproduce it even without injecting a power fault.

5.2 Result Summary

Table 1 summarizes the ACID violations observed under workloads 1–3 (W-1 through W-3) and three different configurations of workload 4 (W-4.1 through W-4.3). W-4.1 uses 2 threads, 10 transactions per thread, each transaction writes to 10 work rows, and the total number of work rows is 1,000. W-4.2 increases the number of threads to 10, the number of work rows being written per transaction to 20, and the total number of work rows to 4,000. W-4.3 further increases the number of work rows written per transaction to 80.

Table 1 shows that 12 out of 15 database/filesystem combinations experienced at least one type of ACID violation under injected power faults. Under relatively simple workloads (i.e., W-1, W-2, and W-3), 11 database/filesystem combinations experienced one or two types of ACID violations. For example, SQL-B violated the C and D properties under workload 3 on both file systems. On the other hand, some databases can handle the power faults better than others. For example, LightningDB does not show any failures under power faults with the first three workloads. Another interesting observation is that power faulting KVS-A causes hangs, preventing the Checker from further execution in a few cases. We cannot access the data and so cannot clearly identify which sort of ACID violation this should be categorized as.

Under the most stressful workload more violations were found. For example, TokyoCabinet violates the atomicity, consistency, and durability properties, and LightningDB violates durability under the most stressful configuration (W-4.3).

The last four columns of Table 1 show for each type of ACID violation the percentage of power faults that cause that violation among all power faults injected under the exhaustive policy, averaged over all workloads. An interesting observation is that the percentage of violations for XFS is always smaller than that for ext3 (except for SQL-B, which shows a similar percentage on both file

DB	FS	W-1	W-2	W-3	W-4.1	W-4.2	W-4.3	A	C	I	D
TokyoCabinet	ext3	D	D	D	A C D	A C D	A C D	0.15	0.14	0	16.05
	XFS	—	D	D	A C D	D	A C D	<0.01	0.01	0	4.38
MariaDB	ext3	D	D	D	D	D	D	0	0	0	1.36
	XFS	D	D	D	D	D	D	0	0	0	0.49
LightningDB	ext3	—	—	—	—	—	D	0	0	0	0.05
	XFS	—	—	—	—	—	—	0	0	0	0
SQLite	ext3	D	D	—	D	D	D	0	0	0	19.15
	XFS	—	—	D	D	D	D	0	0	0	10.60
KVS-A	ext3	—	—	Hang*	—	—	—	0	0	0	0
	XFS	—	—	—	—	—	—	0	0	0	0
SQL-A	ext3	D	D	D	D	D	D	0	0	0	3.31
	XFS	D	D	D	D	D	D	0	0	0	0.92
SQL-B	ext3	D	D	C D	C D	C D	C D	0	8.96	0	3.24
	XFS	C D	D	C D	C D	C D	C D	0	7.77	0	3.90
SQL-C	NTFS	D	D	D	D	D	D	0	0	0	8.08

Table 1: ACID violations observed under workloads 1–3 (W-1 through W-3) and three configurations of workload 4 (W-4.1 through W-4.3). "—" means no failure was observed. The last four columns show for each type of ACID violation the percentage of power faults that cause that violation among all power faults injected under the exhaustive policy, averaged over all workloads. *The checker never reported errors for KVS-A, but in some cases power loss caused a hang in the database code during recovery afterwards. This could potentially be categorized as either an I or a D error; regardless the database is not usable.

systems). This may indicate that XFS is more robust for databases compared to ext3.

Some violations are difficult to trigger. For example, LightningDB violates durability under only 0.05% of the power faults injected under the exhaustive policy. Here the exhaustive approach is not very efficient, and a random sampling approach would be likely to miss the error.

Overall, violation of durability is the most prevalent failure, being found in 7 out of the 8 tested databases and ranging from 0.05% up to 19.15% among all power faults injected. A common type of durability violation is a transaction committed before the power fault being missing after recovery. TokyoCabinet, SQLite, and SQL-B have this failure behavior.

Another common type of durability violation is partial table corruption. Examples include non-retrievable rows, rows retrievable but with corrupted data, or a database crash when touching certain rows. TokyoCabinet, LightningDB, and SQL-B exhibit such failures.

The third type of durability violation is failing to connect to the database upon restart from the post-fault disk image. As a result, the whole table was non-durable. MariaDB, SQL-A, SQL-B, and SQL-C have demonstrated this failure behavior. Our best efforts at manually recovering from this condition failed, except that for MariaDB there is an additional recovery procedure that can allow full recovery. This suggests that for MariaDB the data is likely intact, but the default recovery procedure failed to recognize it upon restart. Although that may be a reasonable strategy for arbitrary image corruption, we do not feel this is completely acceptable behavior under the easy fault model we apply.

5.3 Effectiveness of Patterns

We now evaluate the effectiveness of our pattern-based ranking algorithm at identifying the most vulnerable fault injection points. The five patterns we use (see Section 4.2.2) were extracted based on the ACID violations observed in TokyoCabinet and MariaDB with the exhaustive policy, and we apply them to all 8 tested databases. For SQL-C, we apply only the filesystem-independent and OS-independent patterns (i.e., P_{rep}, P_{jump}, and P_{head}).

Table 2 compares the pattern-based policy with the exhaustive policy under W-4.1 and W-4.3 on ext3 (the results on XFS and under other workloads are similar). Overall, the pattern-based policy is very effective. Injecting power faults at points with scores exceeding 2 can manifest all the types of ACID violations detected by exhaustive testing, except in one or two cases per configuration. For SQL-A, the points with scores exceeding 2 do not suffice; using the score 2 points in addition, however, does suffice to manifest the ACID violations.

The patterns we identified using analysis from only 2 of the databases generalized well to the other 6. Especially for LightningDB, we performed all of the work-

DB	W-4.1		W-4.3	
	match?	top?	match?	top?
TokyoCabinet	Y	Y	Y*	Y
MariaDB	Y	Y	Y	Y
LightningDB	—	—	Y	Y
SQLite	Y	Y	Y	Y
KVS-A	—	—	—	—
SQL-A	Y	N	Y	N
SQL-B	Y	N	Y*	Y
SQL-C	Y	Y	Y	Y

Table 2: Comparison of exhaustive and pattern-based fault injection policies. Y in the match? column means injecting faults at the operations identified by the pattern policy can expose the same types of ACID violations as exposed under the exhaustive policy. Y in the top? column means that power faults need to be injected only at the score 3 or higher points. "—" means no error is detected under both polices. *Extrapolated from a partial run, given the long time frame.

loads, testing, and diagnosis presented in Section 5.1.2 without any further iteration on the framework. Yet we were still able to catch a bug that occurs in only 0.05% of the runs.

5.4 Efficiency of Patterns

We further evaluate the efficiency of our pattern-based policy in terms of the number of fault injection points and the execution time for testing databases with the power faults injected at these points. Table 3 compares the number of fault injection points under the exhaustive and pattern-based policies for W-4.3 on ext3 (the results for the other cases are similar). Under this setting only the points with scores exceeding 2 are needed to manifest the same types of ACID violations as the exhaustive policy (except for SQL-A, which requires points with scores exceeding 1). Compared to the exhaustive policy, the pattern-based policy reduces the number of fault injection points greatly, with an average 21x reduction, while manifesting the same types of ACID violations.

Table 4 further compares the two policies in terms of the execution time required in the replay for testing. Note that the pattern-based policy is very efficient, with an average 19x reduction in execution time compared to the exhaustive policy. For example, we estimate (based on letting it run for 3 days) that exhaustive testing of SQL-B would take over 2 months, while with the pattern-based policy, the testing completed in about 2 days.

DB	Exhaustive	Pattern	%
TokyoCabinet	41,625	7,084	17.0%
MariaDB	1,013	14	1.4%
LightningDB	5,570	171	3.1%
SQLite	438	23	5.3%
KVS-A	4,193	69	1.7%
SQL-A	1,082	53*	4.9%
SQL-B	20,200	936	4.6%
SQL-C	313	2	0.6%
Average	—	—	4.8%

Table 3: Comparison of our two policies in terms of the number of fault injection points under W-4.3. The pattern-based policy includes only points with scores exceeding 2 *except for SQL-A, which includes points with scores exceeding 1.

DB	Exhaustive	Pattern	%
TokyoCabinet	12d 1h*	2d 0h	16.6%
MariaDB	3h 27m	3m 2s	1.5%
LightningDB	7h 56m	20m 44s	4.4%
SQLite	13m 12s	0m 42s	5.3%
KVS-A	5h 17m	5m 32s	1.7%
SQL-A	3h 33m	10m 37s	5.0%
SQL-B	71d 1h*	2d 9h	3.4%
SQL-C	3h 23m	2m 34s	5.1%
Average	—	—	5.4%

Table 4: Comparison of our two policies in terms of replay for testing time under W-4.3. *Estimated based on progress from a 3-day run.

6 Comparison to EXPLODE

EXPLODE [44] is most closely related to our work. It uses ingenious *in situ* model checking to exhaust the state of storage systems to expose bugs, mostly in file systems. Part of our framework is similar: we also use a RAM disk to emulate a block device, and record the resulting trace. However, our fault models are different. EXPLODE simulates a crash model where data may be corrupted before being propagated to disk, and where buffer writes are aggressively reordered. This is quite harsh to upper-level software. Our fault model focuses on faults where the blame more squarely lies on the higher-level software (e.g., databases) rather than on the OS kernel or hardware device. Besides, unlike EXPLODE, we provide explicit suggestions for workloads that are likely to uncover issues in databases, and explicit tracing support to pin found errors to underlying bugs.

When applied to our problem—i.e., testing and diagnosing databases under power fault—EXPLODE has

some limitations. First, the number of manually-defined choice points is limited. The size of the current write set between two choices could be huge, which is especially likely under heavy database transactions. It would be prohibitively expensive, if not impossible, for a model checking tool to exhaust every subset or permutation of the write set on every path. As is, EXPLODE has such a large set of states to explore that in practice it terminates when the user loses patience [44]. Meanwhile, many of the simulated crash images could contain harsh corruption that is unrealistic under power fault. Second, EXPLODE enforces deterministic thread scheduling. This factor, together with the coarse-grained choices, may hide certain types of bugs involving concurrent transactions, a common case in databases. For instance, in a database using `mmap`, the pages maintained in the write set in EXPLODE is likely different from a native run. Specifically, a thread T1 may call `fsync` (which causes the write set to shrink) in its transaction, but due to the enforced scheduling, thread T2 may lose its chance to update the pages before the shrink. Since EXPLODE generates crash images whenever the write set shrinks, it cannot generate an image (and manifest a bug) that requires updates from both T1 and T2. Third, even if a bug is triggered, pinpointing the root cause in a database is still challenging given the code size and complexity. Finally, EXPLODE is built on the Linux kernel, which does not allow testing Windows databases.

On the other hand, by exploring the state space, EXPLODE could exercise different code paths and make corner cases appear as often as common ones, which is a merit of model checking. Thus, we view EXPLODE-like approaches as complementary.

7 Related Work

Testing databases Previous work mostly focuses on functional testing of databases rather than on resilience to external faults. Slutz [38] generates millions of valid SQL queries, and then compares the results from multiple databases; if one database disagrees then that indicates a probable bug. Chays *et al.* [14] proposes a set of tools that automatically generate schema-compliant queries for testing. To improve test coverage, Bati *et al.* [13] propose a genetic approach for generating random test cases for database engines. All of these approaches focus on database bugs in fault-free operation, rather than when power is lost.

Subramanian *et al.* [39] examines the effects of disk corruption on MySQL. Unlike [39], we study the effects of power faults on a range of databases, including closed-source ones, and assume an easy, "perfect" block device under power fault.

Reliability analysis of storage software Similar to EXPLODE [44], MODIST [43] applies model checking to distributed systems and evaluates replicated Berkeley DB. RapiLog [21] analyzes the durability of databases and simplifies the logging by leveraging a formally-verified kernel and synthesized driver. Again, we view these formal methods as complementary. Thanumalayan *et al.* [35] proposes an abstract persistent model (APM) of filesystem properties and studies the effects on application consistency after simulating crashes in the filesystem model. Unlike their modeling approach, we test databases running on real file systems and do not intentionally manipulate the order or content of the blocks. The NoFS [17] shows how a file system can be designed to maintain consistency in the face of crashes; we presume that a database written using the NoFS techniques would be more resilient than those we tested. The IRON file system [36] implements additional redundancy and recovery methods in order to better survive various failures. Again, similar ideas could be applied to databases, although a direct mapping from concepts such as inodes and superblocks to their database equivalents may be nontrivial.

Reliability analysis of storage hardware Several studies have looked at the failure behavior of storage hardware, from spinning magnetic disks [15, 32, 37] to flash memory [19, 20, 40, 46]. Schroeder *et al.* [37] considers in-the-wild failure probabilities, while Chen *et al.* [15] considers how RAID improves the durability and reliability of storage systems. Nightingale *et al.* [32] analyzes hardware failures on PCs including disk subsystem failures. However, none of these studies looks at how the software using the hardware actually responds to faults.

8 Conclusions

We have shown that even ostensibly well-tested databases can lose data. This should be a wake-up call for any author of storage systems software: undirected testing is not enough. Thorough testing requires purpose-built workloads designed to highlight failures, as well as fault injection targeted at those situations in which storage system designers are likely to make mistakes. We can offer no panacea; creating failure-proof storage software is hard. But unless careful attention is paid to correctness, we will continue to cluck our tongues and sigh, while users will continue to cry.

9 Acknowledgments

The authors would like to thank Terrence Kelly, David Andersen (their shepherd), and the anonymous reviewers for their invaluable feedback. This research is partially supported by NSF grants #CCF-0953759 (CAREER Award), #CCF-1218358, and #CCF-1319705, and by a gift from HP.

References

[1] DeWitt Clause. http://en.wikipedia.org/wiki/David_DeWitt.

[2] Discussion on a potential bug on SQLite forum. http://sqlite.1065341.n5.nabble.com/Potential-bug-in-crash-recovery-code-unlink-and-friends-are-not-synchronous-td68885.html.

[3] E2fsprogs: Ext2/3/4 filesystems utilities. http://e2fsprogs.sourceforge.net.

[4] iSCSI wiki. https://en.wikipedia.org/wiki/ISCSI.

[5] LightningDB documents. http://symas.com/mdb/#docs.

[6] Linux SCSI target framework (tgt). http://stgt.sourceforge.net/.

[7] NFI: NTFS file sector information utility. http://support.microsoft.com/kb/253066/en-us/.

[8] PIN — a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool/.

[9] PIN for windows. https://software.intel.com/en-us/articles/pintool-downloads/.

[10] SQLite documents. http://www.sqlite.org/docs.html.

[11] SQLite testing website. http://www.sqlite.org/testing.html.

[12] XFS file system utilities. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Storage_Administration_Guide/xfsothers.html.

[13] BATI, H., GIAKOUMAKIS, L., HERBERT, S., AND SURNA, A. A genetic approach for random testing of database systems. In *Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB '07, VLDB Endowment, pp. 1243–1251.

[14] CHAYS, D., DENG, Y., FRANKL, P. G., DAN, S., VOKOLOS, F. I., AND WEYUKER, E. J. An agenda for testing relational database applications. In *Proceedings of Software Testing, Verification and Reliability* (March 2004), pp. 17–44.

[15] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (June 1994), 145–185.

[16] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1996), ASPLOS VII, ACM, pp. 74–83.

[17] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency Without Ordering. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)* (San Jose, California, February 2012).

[18] CNN. Manufacturer blames super bowl outage on incorrect setting. http://www.cnn.com/2013/02/08/us/superdome-power-outage/, 2013.

[19] GABRYS, R., YAAKOBI, E., GRUPP, L. M., SWANSON, S., AND DOLECEK, L. Tackling intracell variability in TLC flash through tensor product codes. In *ISIT'12* (2012), pp. 1000–1004.

[20] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 24–33.

[21] HEISER, G., LE SUEUR, E., DANIS, A., BUDZYNOWSKI, A., SALOMIE, T.-L., AND ALONSO, G. RapiLog: Reducing System Complexity Through Verification. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 323–336.

[22] HELLERSTEIN, J. M., STONEBRAKER, M., AND HAMILTON, J. Architecture of a database system. *Foundations and Trends in Databases 1*, 2 (2007), 141–259.

[23] IBM. Database logging using raw devices is deprecated. http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.rn.doc/doc/c0023086.htm, 2006.

[24] LEACH, A. Level 3's UPS burnout sends websites down in flames. http://www.theregister.co.uk/2012/07/10/data_centre_power_cut/, 2012.

[25] MCMILLAN, R. Amazon blames generators for blackout that crushed Netflix. http://www.wired.com/wiredenterprise/2012/07/amazon_explains/, 2012.

[26] MICROSOFT. NTFS vs. FAT vs. raw partitions. http://technet.microsoft.com/en-us/library/cc966414.aspx, 2007.

[27] MILLER, R. Human error cited in hosting.com outage. http://www.datacenterknowledge.com/archives/2012/07/28/human-error-cited-hosting-com-outage/, 2012.

[28] MILLER, R. Power outage hits London data center. http://www.datacenterknowledge.com/archives/2012/07/10/power-outage-hits-london-data-center/, 2012.

[29] MILLER, R. Data center outage cited in visa downtime across canada. http://www.datacenterknowledge.com/archives/2013/01/28/data-center-outage-cited-in-visa-downtime-across-canada/, 2013.

[30] MILLER, R. Power outage knocks DreamHost customers offline. http://www.datacenterknowledge.com/archives/2013/03/20/power-outage-knocks-dreamhost-customers-offline/, 2013.

[31] NBC. Power outage affects thousands, including columbus hospital. http://columbus.gotnewswire.com/news/power-outage-affects-thousands-including-columbus-hospital, 2014.

[32] NIGHTINGALE, E. B., DOUCEUR, J. R., AND ORGOVAN, V. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 343–356.

[33] ORACLE. Desupport raw storage device. http://docs.oracle.com/cd/E16655_01/server.121/e17642/deprecated.htm, 2013.

[34] PARK, S., KELLY, T., AND SHEN, K. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 225–238.

[35] PILLAI, T. S., CHIDAMBARAM, V., ALAGAPPAN, R., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)* (October 2014).

[36] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)* (Brighton, United Kingdom, October 2005), pp. 206–220.

[37] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)* (2007).

[38] SLUTZ, D. R. Massive stochastic testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1998), VLDB '98, Morgan Kaufmann Publishers Inc., pp. 618–622.

[39] SUBRAMANIAN, S., ZHANG, Y., VAIDYANATHAN, R., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND NAUGHTON, J. F. Impact of disk corruption on open-source DBMS. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on* (2010), IEEE, pp. 509–520.

[40] TSENG, H.-W., GRUPP, L. M., AND SWANSON, S. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference (DAC'11)* (2011).

[41] VERGE, J. Internap data center outage takes down Livestream and StackExchange. http://www.datacenterknowledge.com/archives/2014/05/16/internap-data-center-outage-takes-livestream-stackexchange/, 2014.

[42] WOLFFRADT, R. S. V. Fire in your data center: No power, no access, now what? http://www.govtech.com/state/Fire-in-your-Data-Center-No-Power-No-Access-Now-What.html, 2014.

[43] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2009), NSDI'09, pp. 213–228.

[44] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)* (November 2006), pp. 131–146.

[45] ZELLER, A. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (New York, NY, USA, 2002), SIGSOFT '02/FSE-10, ACM, pp. 1–10.

[46] ZHENG, M., TUCEK, J., QIN, F., AND LILLIBRIDGE, M. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)* (2013).

Fast Databases with Fast Durability and Recovery Through Multicore Parallelism

Wenting Zheng, *MIT** Stephen Tu, *MIT**
Eddie Kohler, *Harvard University* Barbara Liskov, *MIT*

Abstract

Multicore in-memory databases for modern machines can support extraordinarily high transaction rates for online transaction processing workloads. A potential weakness, however, is recovery from crash failures. Can classical techniques, such as checkpoints, be made both efficient enough to keep up with current systems' memory sizes and transaction rates, and smart enough to avoid additional contention? Starting from an efficient multicore database system, we show that naive logging and checkpoints make normal-case execution slower, but that frequent disk synchronization allows us to keep up with many workloads with only a modest reduction in throughput. We design throughout for parallelism: during logging, during checkpointing, and during recovery. The result is fast. Given appropriate hardware (three SSDs and a RAID), a 32-core system can recover a 43.2 GB key-value database in 106 seconds, and a $>$ 70 GB TPC-C database in 211 seconds.

1 Introduction

In-memory databases on modern multicore machines [10] can handle complex, large transactions at millions to tens of millions of transactions per second, depending on transaction size. A potential weakness of such databases is robustness to crashes and power failures. Replication can allow one site to step in for another, but even replicated databases must write data to persistent storage to survive correlated failures, and performance matters for both persistence and recovery.

Crash resistance mechanisms, such as logging and checkpointing, can enormously slow transaction execution if implemented naively. Modern fast in-memory databases running tens of millions of small transactions per second can generate more than 50 GB of log data per minute when logging either values or operations. In terms of both transaction rates and log sizes, this is up to several orders of magnitude more than the values reported in previous studies of in-memory-database durability [2, 14, 24]. Logging to disk or flash is at least theoretically fast, since log writes are sequential, but sequential log replay is not fast on a modern multicore machine. Checkpoints are also required, since without them, logs would grow without bound, but checkpoints require a walk over the entire database, which can cause data movement and cache pollution that reduce concurrent transaction performance. Recovery of a multi-gigabyte database using a single core could take more than 90 minutes on today's machines, which is a long time even in a replicated system.

Our goal in this work was to develop an in-memory database with full persistence at relatively low cost to transaction throughput, and with fast recovery, meaning we hoped to be able to recover a large database to a transactionally-consistent state in just a few minutes without replication. Starting from Silo [27], a very fast in-memory database system, we built *SiloR*, which adds logging, checkpointing, and recovery. Using a combination of logging and checkpointing, we are able to recover a 43.2 GB YCSB key-value-style database to a transactionally-consistent snapshot in 106 seconds, and a more complex $>$ 70 GB TPC-C database with many tables and secondary indexes in 211 seconds.

Perhaps more interesting than our raw performance is the way that performance was achieved. We used concurrency in all parts of the system. The log is written concurrently to several disks, and a checkpoint is taken by several concurrent threads that also write to multiple disks. Concurrency was crucial for recovery, and we found that the needs of recovery drove many of our design decisions. The key to fast recovery is using all of the machine's resources, which, on a modern machine, means using all cores. But some designs tempting on the logging side, such as operation logging (that is, logging transaction types and arguments rather than logging values), are difficult to recover in parallel. This drive for fast parallel recovery affected many aspects of our logging and checkpointing designs.

Starting with an extremely fast in-memory database, we show:

- All the important durability mechanisms can and should be made parallel.
- Checkpointing can be fast without hurting normal transaction execution. The fastest checkpoints introduce undesired spikes and crashes into concurrent throughput, but through good engineering and by pacing checkpoint production, this variability can be reduced enormously.

*Currently at University of California, Berkeley.

- Even when checkpoints are taken frequently, a high-throughput database will have to recover from a very large log. In our experiments, log recovery is the bottleneck; for example, to recover a 35 GB TPC-C database, we recover 16 GB from a checkpoint and 180 GB from the log, and log recovery accounts for 90% of recovery time. Our design allows us to accomplish log replay at roughly the maximum speed of I/O.

- The system built on these ideas can recover a relatively large database quite quickly.

2 Silo overview

We build on Silo, a fast in-memory relational database that provides tables of typed records. Clients issue one-shot requests: all parameters are available when a request begins, and the request does not interact with its caller until it completes. A request is dispatched to a single database *worker* thread, which carries it out to completion (commit or abort) without blocking. Each worker thread is pinned to a physical core of the server machine. Most cores run workers, but SiloR reserves several cores for logging and checkpointing tasks.

Silo tables are stored in efficient, cache-friendly concurrent B-trees [15]. Each table uses one primary tree and zero or more secondary trees for secondary indexes. Key data is embedded in tree structures, and values are stored in separately-allocated *records*. All structures are stored in shared memory, so any worker can access the entire database.

Silo uses a variant of optimistic concurrency control (OCC) [11] to serialize transactions. Concurrency control centers on *transaction IDs* (TIDs). Each record contains the TID of the transaction that most recently modified it. As a worker runs a transaction, it maintains a *read-set* containing the old TID of each read or written record, and a *write-set* containing the new state of each written record. On transaction completion, a worker determines whether the transaction can commit. First it locks the records in the write-set (in a global order to avoid deadlock). Then it computes the transaction's TID; this is the serialization point. Next it compares the TIDs of records in the read-set with those records' current TIDs, and aborts if any TIDs have changed or any record is locked by a different transaction. Otherwise it commits and overwrites the write-set records with their new values and the new TID.

2.1 Epochs

Silo transaction IDs differ in an important way from those in other systems, and this difference impacts the way SiloR does logging and recovery. Classical OCC obtains the TID for a committing transaction by effectively incrementing a global counter. On modern multicore hardware, though, any global counter can become a source of performance-limiting contention. Silo eliminates this contention using time periods called *epochs* that are embedded in TIDs. A global epoch number E is visible to all threads. A designated thread advances it periodically (every 40 ms). Worker threads use E during the commit procedure to compute the new TID. Specifically, the new TID is (a) greater than any TID in the read-set, (b) greater than the last TID committed by this worker, and (c) in epoch E.

This avoids false contention on a global TID, but fundamentally changes the relationship between TIDs and the serial order. Consider concurrent transactions T1 and T2 where T1 reads a key that T2 then overwrites. The relationship between T1 and T2 is called an *anti-dependency*: T1 must be ordered before T2 because T1 depends on *the absence* of T2. In conventional OCC, whose TIDs capture anti-dependencies, our example would always have TID(T1) $<$ TID(T2). But in Silo, there is no communication whatsoever from T1 to T2, and we could find TID(T1) $>$ TID(T2)! This means that replaying a Silo database's committed transactions in TID order might recover the wrong database.

Epochs provide the key to correct replay. On total-store-order (TSO) architectures like x86-64, the designated thread's update of E becomes visible at all workers simultaneously. Because workers read the current epoch at the serialization point, the ordering of TIDs *with different epochs* is always compatible with the serial order, even in the case of anti-dependencies. Epochs allow for a form of group commit: SiloR persists and recovers in units of epochs. We describe below how this impacts logging, checkpointing, and recovery.

3 Logging

This section explains how SiloR logs transaction modifications for persistence. Our design builds on Silo, which included logging but did not consider recovery, log truncation, or checkpoints. The SiloR logging subsystem adds log truncation, makes changes related to liveness, and allows more parallelism on replay.

3.1 Basic logging

The responsibility for logging in SiloR is split between workers, which run transactions, and separate logging threads ("loggers"), which handle only logging, checkpointing, and other housekeeping tasks. Workers generate log records as they commit transactions; they pass these records to loggers, which commit the logs to disk. When a set of logs is committed to disk via fsync, the loggers inform the workers. This allows workers to send transaction results to clients.

A log record comprises a committed transaction's TID plus the table, key, and value information for all records modified by that transaction. Each worker constructs log records in disk format and stores them in a memory buffer taken from a per-worker buffer pool. When a buffer fills, or at an epoch boundary, the worker passes the buffer to the logger over a shared-memory queue.

3.2 Value logging vs. operation logging

SiloR uses value logging, not operation or transaction logging. This means that SiloR logs contain each transaction's output keys and values, rather than the identity of the executed operation and its parameters.

The choice of value logging is an example of recovery parallelism driving the normal-case logging design. Value logging has an apparent disadvantage relative to operation logging: for many workloads (such as TPC-C) it logs more data, and therefore might unnecessarily slow transaction execution. However, from the point of view of recovery parallelism, the advantages of value logging outweigh its disadvantages. Value logging is easy to replay in parallel—the largest TID per value wins. This works in SiloR because TIDs reflect dependencies, i.e., the order of writes, and because we recover in units of epochs, ensuring that anti-dependencies are not a problem. Operation logging, in contrast, requires that transactions be replayed in their original serial order. This is always hard to parallelize, but in Silo, it would additionally require logging read-sets (keys and TIDs) to ensure anti-dependencies were obeyed. Operation logging also requires that the initial pre-replay database state be a transactionally consistent snapshot, which value logging does not; and for small transactions value and operation logs are about the same size. These considerations led us to prefer value logging in SiloR. We solve the problem of value logging I/O by adding hardware until logging is not a bottleneck, and then using that hardware wisely.

3.3 Workers and loggers

Loggers have little CPU work to do. They collect logs from workers, write them to disk, and await durability notification from the kernel via the fsync/fdatasync system call. Workers, of course, have a lot of CPU work to do. A SiloR deployment therefore contains many worker threads and few logger threads. We allocate enough logger threads per disk to keep that disk busy, one per disk in our evaluation system.

But how should worker threads map to logger threads? One possibility is to assign each logger a partition of the database. This might reduce the data written by loggers (for example, it could improve the efficacy of compression), and it might speed up replay. We rejected this design because of its effect on normal-case transaction execution. Workers would have to do more work to analyze transactions and split their updates appropriately. More fundamentally, every worker might have to communicate with every logger. Though log records are written in batches (so the communication would not likely introduce contention), this design would inevitably introduce *remote writes or reads*: physical memory located on one socket would be accessed, either for writes or reads, by a thread running on a different socket. Remote accesses are expensive and should be avoided when possible.

Our final design divides workers into disjoint subsets, and assigns each subset to exactly one logger. Core pinning is used to ensure that a logger and its workers run on the same socket, making it likely that log buffers allocated on a socket are only accessed by that socket.

3.4 Buffer management

Although loggers should not normally limit transaction execution, loggers must be able to apply backpressure to workers, so that workers don't generate indefinite amounts of log data. This backpressure is implemented by buffer management. Loggers allocate a maximum number of log buffers per worker core. Buffers circulate between loggers and workers as transactions execute, and a worker blocks when it needs a new log buffer and one is not available. A worker flushes a buffer to its logger when either the buffer is full or a new epoch begins, whichever comes first. It is important to flush buffers on epoch changes, whether or not those buffers are full, because SiloR cannot mark an epoch as persistent until it has durably logged *all* transactions that happened in that epoch. Each log buffer is 512 KB. This is big enough to obtain some benefit from batching, but small enough to avoid wasting much space when a partial buffer is flushed.

We found that log-buffer backpressure in Silo triggered unnecessarily often because it was linked with fsync times. Loggers amplified file system hiccups, such as those caused by concurrent checkpoints, into major dips in transaction rates. SiloR's loggers instead recirculate log buffers back to workers as soon as possible—after a write, rather than after the following epoch change and fsync. We also increased the number of log buffers available to workers, setting this to about 10% of the machine's memory. The result was much less noise in transaction execution rates.

3.5 File management

Each SiloR logger stores its log in a collection of files in a single directory. New entries are written to a file called **data.log**, the current log file. Periodically (currently every 100 epochs) the logger renames this file to **old_data.*e***, where *e* is the largest epoch the file contains, then starts a new **data.log**. Using multiple files simplifies the process of log truncation and, in our measure-

ments, didn't slow logging relative to Silo's more primitive single-file design.

Log files do not contain transactions in serial order. A log file contains concatenated log buffers from several workers. These buffers are copied into the log without rearrangement; in fact, to reduce data movement, SiloR logger threads don't examine log data at all. A log file can even contain *epochs* out of order: a worker that delays its release of the previous epoch's buffer will not prevent other workers from producing buffers in the new epoch. All we know is that a file **old_data.***e* contains no records with epochs $> e$. And, of course, a full log comprises multiple log directories stored independently by multiple loggers writing to distinct disks. Thus, no single log contains enough information for recovery to produce a correct database state. It would be possible to extract this information from *all* logs, but instead SiloR uses a distinguished logger thread to maintain another file, **pepoch**, that contains the current *persistent epoch*. The logger system guarantees that all transactions in epochs $\leq$ **pepoch** are durably stored in some log. This epoch is calculated as follows:

1. Each worker w advertises its *current* epoch, e_w, and guarantees that all future transactions it sends to its logger will have epoch $\geq e_w$. It updates e_w by setting $e_w \leftarrow E$ after flushing its current log buffer to its logger.

2. Each logger l reads log buffers from workers and writes them to log files.

3. Each logger regularly decides to make its writes durable. At that point, it calculates the minimum of the e_w for each of its workers and the epoch number of any log buffer it owns that remains to be written. This is the logger's current epoch, e_l. The logger then synchronizes all its writes to disk.

4. After this synchronization completes, the logger publishes e_l. This guarantees that all associated transactions with epoch $< e_l$ have been durably stored for this logger's workers.

5. The distinguished logger thread periodically computes a *persistence* epoch e_p as $\min\{e_l\} - 1$ over all loggers. It writes e_p to the **pepoch** file and then synchronizes that write to disk.

6. Once **pepoch** is durably stored, the distinguished logger thread publishes e_p to a global variable. At that point all transactions with epochs $\leq e_p$ have become durable and workers can release their results to clients.

This protocol provides a form of group commit. It ensures that the logs contain all information about transactions in epochs $\leq e_p$, and that no results from transactions with epoch $> e_p$ were released to clients. Therefore it is safe for recovery to recover all transactions with epochs $\leq e_p$, and also necessary since those results may have been released to clients. It has one important disadvantage, namely that the critical path for transaction commit contains two fsyncs (one for the log file and one for **pepoch**) rather than one. This somewhat increases latency.

4 Checkpoints

Although logs suffice to recover a database, they do not suffice to recover a database in bounded time. In-memory databases must take periodic *checkpoints* of their state to allow recovery to complete quickly, and to support log truncation. This section describes how SiloR takes checkpoints.

4.1 Overview

Our main goal in checkpoint production is to produce checkpoints as quickly as possible without disrupting worker throughput. Checkpoint speed matters because it limits the amount of log data that will need to be replayed at recovery. The smaller the distance between checkpoints, the less log data needs to be replayed, and we found the size of the log to be the major recovery expense. Thus, as with log production, checkpointing uses multiple threads and multiple disks.

Checkpoints are written by *checkpointer* threads, one per checkpoint disk. In our current implementation checkpoints are stored on the same disks as logs, and loggers and checkpointers execute on the same cores (which are separate from the worker cores that execute transactions). Different checkpointers are responsible for different slices of the database; a distinguished *checkpoint manager* assigns slices to checkpointers. Each checkpointer's slices amount to roughly $1/n$th of the database, where n is the number of disks. A checkpoint is associated with a range of epochs $[e_l, e_h]$, where each checkpointer started its work during or after e_l and finished its work during or before e_h.

Each checkpointer walks over its assigned database slices in key order, writing records as it goes. Since OCC installs modifications at commit time, all records seen by checkpointers are committed. This means that full ARIES-style undo and redo logging is unnecessary; the log can continue to contain only "redo" records for committed transactions. However, concurrent transactions continue to execute during the checkpoint period, and they do not coordinate with checkpointers except via per-record locks. If a concurrent transaction commits multiple modifications, there is no guarantee the checkpointers will see them all. SiloR checkpoints are thus inconsistent or "fuzzy": the checkpoint is not necessarily

a consistent snapshot of the database as of a particular point in the serial order. To recover a consistent snapshot, it is always necessary both to restore a checkpoint and to replay at least a portion of the log.

We chose to produce an inconsistent checkpoint because it's less costly in terms of memory usage than a consistent checkpoint. Silo could produce consistent checkpoints using its support for snapshot transactions [27]. However, checkpoints of large databases take a long time to write (multiple tens of seconds), which is enough time for all database records to be overwritten. The memory expense associated with preserving the snapshot for this period, and especially the allocation expense associated with storing new updates in newly-allocated records (rather than overwriting old records), reduces normal-case transaction throughput by 10% or so. We prefer better normal-case throughput. Our choice of inconsistent checkpoints further necessitates our choice of value logging; it is impossible to recover from an inconsistent checkpoint without either value logging or some sort of ARIES-style undo logging.

Another possible design for checkpoints is to avoid writing information about keys whose records haven't changed since the previous checkpoint, for example, designing a disk format that would allow a new checkpoint to elide unmodified key ranges. We rejected this approach because ours is simpler, and also because challenging workloads, such as uniform updates, can cause any design to effectively write a complete checkpoint every time a checkpoint is required. We wanted to understand the performance limits caused by these workloads.

In an important optimization, checkpointer threads skip any records with current epoch $\geq e_l$. Thus, the checkpoint contains those keys written in epochs $< e_l$ that were not overwritten in epochs $\geq e_l$. It is not necessary to write such records because, given any inconsistent checkpoint started in e_l, it is always necessary to replay the log starting at epoch e_l. Specifically, the log must be complete over a range of epochs $[e_l, e_x]$, where $e_x \geq e_h$, for recovery of a consistent snapshot to be possible. There's no need to store a record in the checkpoint that will be replayed by the log. This optimization reduces our checkpoint sizes by 20% or more.

4.2 Writing the checkpoint

Checkpointers walk over index trees to produce the checkpoint. Since we want each checkpointer to be responsible for approximately the same amount of work, yet tables differ in size, we have all checkpointers walk over all tables. To make the walk efficient, we partition the keys of each table into n subranges, one per checkpointer. This way each checkpointer can take advantage of the locality for keys in the tree.

The checkpoint is organized to enable efficient recovery. During recovery, *all* cores are available, so we designed the checkpoint to facilitate using those cores.

For each table, each checkpointer divides its assigned key range into m files, where m is the number of cores that would be used during recovery for that key range. Each of a checkpointer's m files are stored on the same disk. As the checkpointer walks over its range of the table, it writes blocks of keys to these m files. Each block contains a contiguous range of records, but blocks are assigned to files in round-robin order. There is a tension here between two aspects of fast recovery. On the one hand, recovery is more efficient when a recovery worker is given a continuous range of records, but on the other hand, recovery resources are more effectively used when the recovery workload is evenly distributed (each of the m files contain about the same amount of work). Calculating a perfect partition of an index range into equal-size subranges is somewhat expensive, since to do this requires tree walks. We chose a point on this tradeoff where indexes are coarsely divided among checkpointers into roughly-equal subranges, but round-robin assignment of blocks to files evens the workload at the file level.

The checkpoint manager thread starts a new checkpoint every C seconds. It picks the partition for each table and writes this information into a shared array. It then records e_l, the checkpoint's starting epoch, and starts up n checkpointer threads, one per disk. For each table, each thread creates the corresponding checkpoint files and walks over its assigned partition using a range scan on the index tree. As it walks, it constructs a block of record data, where each record is stored as a key/TID/value tuple. When its block fills up, the checkpointer writes that block to one of the checkpoint files and continues. The next full block is written to the next file in round-robin order.

Each time a checkpointer's outstanding writes exceed 32 MB, it syncs them to disk. These intermediate syncs turned out to be important for performance, as we discuss in §6.2.

When a checkpointer has processed all tables, it does a final sync to disk. It then reads the current epoch E and reports this information to the manager. When all checkpointers have reported, the manager computes e_h; this is the maximum epoch reported by the checkpointers, and thus is the largest epoch that might have updates reflected in the checkpoint. Although, thanks to our reduced checkpoint strategy, new tuples created during e_h are not stored in the checkpoint, tuples removed or overwritten during e_h are also not stored in the checkpoint, so the checkpoint can't be recovered correctly without complete logs up to and including e_h. Thus, the manager waits until $e_h \leq e_p$, where e_p is the persistence

epoch computed by the loggers (§3.5). Once this point is reached, the manager *installs* the checkpoint on disk by writing a final record to a special checkpoint file. This file records e_l and e_h, as well as checkpoint metadata, such as the names of the database tables and the names of the checkpoint files.

4.3 Cleanup

After the checkpoint is complete, SiloR removes old files that are no longer needed. This includes any previous checkpoints and any log files that contain only transactions with epochs $< e_l$. Recall that each log comprises a current file and a number of earlier files with names like **old_data**.e. Any file with $e < e_l$ can be deleted.

The next checkpoint is begun roughly 10 seconds after the previous checkpoint completed. Log replay is far more expensive than checkpoint recovery, so we aim to minimize log replay by taking frequent checkpoints. In future work, we would like to investigate a more flexible scheme that, for example, could delay a checkpoint if the log isn't growing too fast.

5 Recovery

SiloR performs recovery by loading the most recent checkpoint, then correcting it using information in the log. In both cases we use many concurrent threads to process the data and we overlap processing and I/O.

5.1 Checkpoint recovery

To start recovery, a recovery manager thread reads the latest checkpoint metadata file. This file contains information about what tables are in the system and e_l, the epoch in which the checkpoint started. The manager creates an in-memory representation for each of the T index trees mentioned in the checkpoint metadata. In addition it deletes any checkpoint files from earlier or later checkpoints and removes all log files from epochs before e_l.

The checkpoint is recovered concurrently by many threads. Recall that the checkpoint consists of many files per database table. Each table is recorded on all n disks, partitioned so that on each disk there are m files for each table. Recovery is carried out by $n \times m$ threads. Each thread reads from one disk, and is responsible for reading and processing T files from that disk (one file per index tree). Processing is straightforward: for each key/value/TID in the file, the key is inserted in the index tree identified by the file name, with the given value and TID. Since the files contain different key ranges, checkpoint recovery threads are able to reconstruct the tree in parallel with little interference; additionally they benefit from locality when processing a subrange of keys in a particular table.

5.2 Log recovery

After all threads have finished their assigned checkpoint recovery tasks, the system moves on to log recovery. As mentioned in §3, there was no attempt at organizing the log records at runtime (e.g. partitioning the log records based on what tables were being modified). Instead it is likely that each log file is a jumble of modifications to various index trees. This situation is quite different than it was for the checkpoint, which was organized so that concurrent threads could work on disjoint partitions of the database. However, SiloR uses value logging, which has the property that the logs can be processed in any order. All we require is that at the end of processing, every key has an associated value corresponding to the last modification made up through the most recent persistent epoch prior to the failure. If there are several modifications to a particular key k, these will have associated TIDs T1, T2, and so on. Only the entry with the largest of these TIDs matters; whether we happen to find this entry early or late in the log recovery step does not.

We take advantage of this property to process the log in parallel, and to avoid unnecessary allocations, copies, and work. First the manager thread reads the **pepoch** file to obtain e_p, the number of the most recent persistent epoch. All log records for transactions with TIDs for later epochs are ignored during recovery. This is important for correctness since group commit has not finished for those later epochs; if we processed records for epochs after e_p we could not guarantee that the resulting database corresponded to a prefix of the serial order.

The manager reads the directory for each disk, and creates a variable per disk, L_d, that is used to track which log files from that disk have been processed. Initially this variable is set to the number of relevant log files for that disk, which, in our experiments, is in the hundreds. Then the manager starts up g log processor threads for each disk. We use all threads during log recovery. For instance, on a machine with N cores and n disks, we have $g = \lceil N/n \rceil$. This can produce more recovery threads than there are cores. We experimented with the alternative $m = \lfloor N/n \rfloor$, but this leaves some cores idle during recovery, and we observed worse recovery times than with oversubscription.

A log processor thread proceeds as follows. First it reads, decrements, and updates L_d for its disk. This update is done atomically: this way it learns what file it should process, and updates the variable so that the next log processor for its disk will process a different file. If the value it reads from L_d is ≤ 0, the log processor thread has no more work to do. It communicates this to the manager and stops. Otherwise the processor thread reads the next file, which is the *newest* file that has not yet been processed. In other words, we process the files in the opposite order than they were written. The proces-

sor thread on disk d that first reads L_d processes the current log file **data.log**; after this files are read in reverse order by the epoch numbers contained in their names. The files are large enough that, when reading them, we get good throughput from the disk; there's little harm in reading the files out of order (i.e., in an order different from the order they were written).

The processor thread reads the entries in the file sequentially. Recall that each entry contains a TID t and a set of table/key/value tuples. If t contains an epoch number that is $< e_l$ or $> e_p$, the thread skips the entry. Otherwise, the thread inserts a record into the table if its key isn't there yet; when a version of the record is already in the table, the thread overwrites only if the log record has a larger TID.

Value logging replay has the same result no matter what order files are processed. We use reverse order for reading log files because it uses the CPU more efficiently than forward order when keys are written multiple times. When files are processed in strictly forward order, every log record will likely require overwriting some value in the tree. When files are processed in roughly reverse order, and keys are modified multiple times, then many log records don't require overwriting: the tree's current value for the key, which came from a later log file, is often newer than the log record.

5.3 Correctness

Our recovery strategy is correct because it restores the database to the state it had at the end of the last persistent epoch e_p. The state of the database after processing the checkpoint is definitely not correct: it is inconsistent, and it is also missing modifications of persistent transactions that ran after it finished. All these problems are corrected by processing the log. The log contains all modifications made by transactions that ran in epochs in e_l up through e_p. Therefore it contains what is needed to rectify the checkpoint. Furthermore, the logic used to do the rectification leads to each record holding the modification of the last transaction to modify it through epoch e_p, because we make this decision based on TIDs. And, importantly, we ignore log entries for transactions from epochs after e_p.

It's interesting to note that value logging works without having to know the exact serial order. All that is required is enough information so that we can figure out the most recent modification. That is, log record "version numbers" must capture dependencies, but need not capture anti-dependencies. Silo TIDs meet this requirement. And because TID comparison is a simple commutative test, log processing can take place in any order. In addition, of course, we require the group commit mechanism provided by epochs to ensure that anti-dependencies are also preserved.

6 Evaluation

In this section, we evaluate the effectiveness of the techniques in SiloR, confirming the following performance hypotheses:

- SiloR's checkpointer has only a modest effect on both the latency and throughput of transactions on a challenging write-heavy key-value workload and a typical online transaction processing workload.
- SiloR recovers 40–70 GB databases within minutes, even when crashes are timed to maximize log replay.

6.1 Experimental setup

All of our experiments were run on a single machine with four 8-core Intel Xeon E7-4830 processors clocked at 2.1 GHz, yielding a total of 32 physical cores. Each core has a private 32 KB L1 cache and a private 256 KB L2 cache. The eight cores on a single processor share a 24 MB L3 cache. The machine has 256 GB of DRAM with 64 GB of DRAM attached to each socket, and runs 64-bit Linux 3.2.0. We run our experiments without networked clients; each database worker thread runs with an integrated workload generator. We do not take advantage of our machine's NUMA-aware memory allocator, a decision discussed in §6.5.

We use three separate Fusion ioDrive2 flash drives and one RAID-5 disk array. Each disk is used for both logging and checkpointing. Each drive has a dedicated logger thread and checkpointer thread, both of which run on the same core. Within a drive, the log and checkpoint information reside in separate files. Each logger or checkpointer writes to a series of files on a single disk.

We measure three related databases, SiloR, LogSilo, and MemSilo. These systems have identical in-memory database structures. SiloR is the full system described here, including logging and checkpointing. LogSilo is a version of SiloR that only logs data: there are no checkpointer threads or checkpoints. MemSilo is Silo run without persistence, and is a later version of the system of Tu et al. [27] Unless otherwise noted, we run SiloR and LogSilo with 28 worker threads and MemSilo with 32 worker threads.

6.2 Key-value workload

To demonstrate that SiloR can log and checkpoint with low overhead, we run SiloR on a variant of YCSB workload mix A. YCSB is a popular key-value benchmark from Yahoo [4]. We modified YCSB-A to have a read/write (get/put) ratio of 70/30 (not 50/50), and a record size of 100 bytes (not 1000). This workload mix was originally designed for MemSilo to stress database internals rather than memory allocation; though the read/write ratio is somewhat less than standard YCSB-A, it is still quite high compared to most workloads. Our read and write transactions sample keys uniformly.

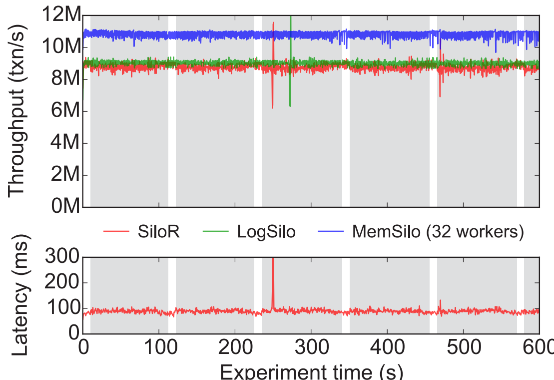


Figure 1: Throughput and latency of SiloR, and throughput of LogSilo and MemSilo, on our modified YCSB benchmark. Average throughput was 8.76 Mtxn/s, 9.01 Mtxn/s, and 10.83 Mtxn/s, respectively. Average SiloR latency was 90 ms/txn. Database size was 43.2 GB. Grey regions show those times when the SiloR experiment was writing a checkpoint.

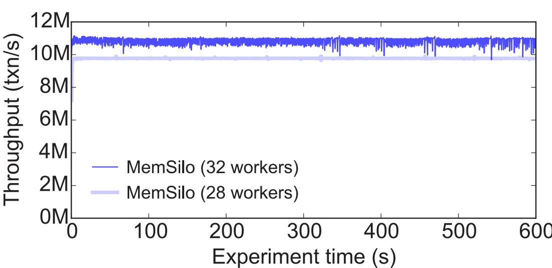


Figure 2: Throughput of MemSilo on YCSB with 32 and 28 workers. Average throughput was 10.83 Mtxn/s and 9.77 Mtxn/s, respectively.

There are 400M keys for a total database size of roughly 43.2 GB (3.2 GB of key data, 40 GB of value data).

Figure 1 shows the results over a 10-minute experiment. Checkpointing can be done concurrently with logging without greatly affecting transaction throughput. The graph shows, over the length of the experiment, rolling averages of throughput and latency with a 0.5-second averaging window. For SiloR and LogSilo, throughput and latency are measured to transaction persistence (i.e., latency is from the time a transaction is submitted to the time SiloR learns the transaction's effects are persistent). Intervals during which the checkpointer is running are shown in gray. Figure 1's results are typical of our experimental runs; Figure 6 in the appendix shows two more runs.

SiloR is able to run multiple checkpoints and almost match LogSilo's throughput. Its throughput is also close to that of MemSilo, although MemSilo does no logging or checkpointing whatsoever: SiloR achieves

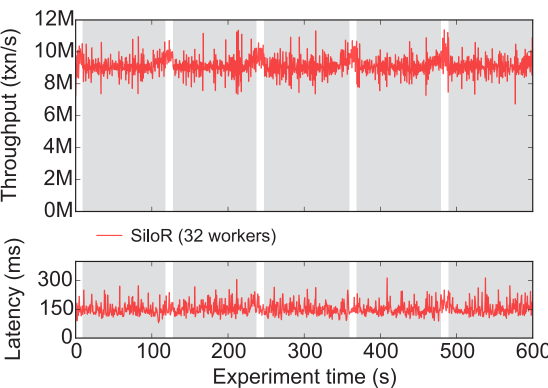


Figure 3: Throughput and latency of SiloR on YCSB with 32 workers. Average throughput was 9.14 Mtxn/s and average latency 153 ms.

8.76 Mtxn/s, 80% the average throughput of MemSilo (10.83 Mtxn/s). Average latency is affected by logging and checkpointing somewhat more significantly; it is 90 ms/transaction.[1] Some of this latency is inherent in Silo's epoch design. Since the epoch advances every 40 ms, average latency cannot be less than 20 ms. The rest is due to a combination of accumulated batching delays (workers batch transactions in log buffers, loggers batch updates to synchronizations) and delays in the persistent storage itself (i.e., the two fsyncs in the critical path each take 10–20 ms, and sometimes more). Nevertheless, we believe this latency is not high for a system involving persistent storage.

During the experiment, SiloR generates approximately 298 MB/s of IO per disk. The raw bandwidth of our Fusion IO drives is reported as 590 MB/s/disk; we are achieving roughly half of this.

SiloR and LogSilo's throughput is less than MemSilo's for several reasons, but as Figure 2 shows, an important factor is simply that MemSilo has more workers available to run transactions. SiloR and LogSilo require extra threads to act as loggers and checkpointers; we run four fewer workers to leave cores available for those threads. If we run MemSilo with 28 workers, its throughput is reduced by roughly 10% to 9.77 Mtxn/s, making up more than half the gap with SiloR. We also ran SiloR with 32 workers. This bettered the average throughput to 9.13 Mtxn/s, but CPU oversubscription caused wide variability in throughput and latency (Figure 3).

As we expect, the extensive use of group commit in LogSilo and SiloR make throughput, and particularly latency, more variable than in MemSilo. Relative to Mem-

[1] Due to a technical limitation in SiloR's logger implementation, the latency shown in the figure is the (running) average latency for *write* transactions only; we believe these numbers to be a converative upper bound on the actual latency of the system.

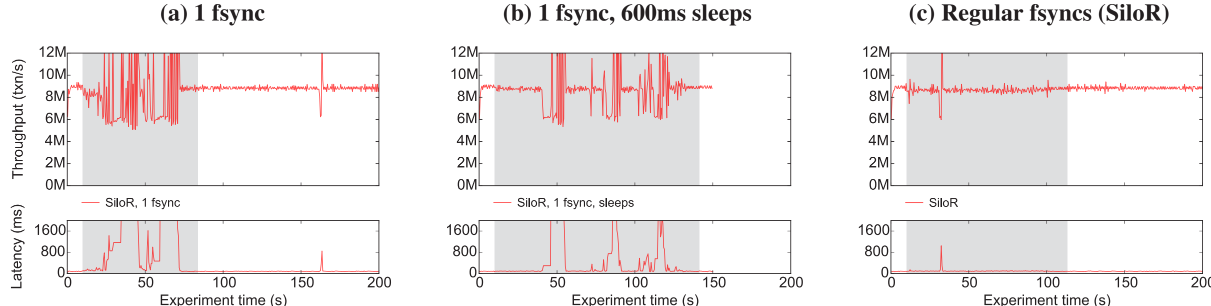


Figure 4: Importance of regular disk synchronization. In (a), one fsync call synchronizes the checkpoint; throughput and latency are extremely bursty (note the latency axis tops out at 2 sec). In (b), regular sleep calls in the checkpoint threads reduce burstiness, but do not eliminate it. In (c), SiloR, regular calls to fsync almost entirely eliminate burstiness. Here we run the modified YCSB benchmark.

Silo with 28 cores, LogSilo's performance is more variable, and SiloR's more variable still. The spike in latency visible in Figure 1, which happened at one time or another in most of our runs, is discussed below in §6.5.

Importance of regular synchronization. A checkpoint is useless until it is complete, so the obvious durability strategy for a checkpointer thread is to call fsync once, after writing all checkpoint data and before reporting completion to the manager. But SiloR checkpointers call fsync far more frequently—once per 32 MB of data written. Figure 4 shows why this matters: the naive strategy, (a), is very unstable on our Linux system, inducing wild throughput swings and extremely high latency. Slowing down checkpointer threads through the occasional introduction of sleep() calls, (b), reduces the problem, but does not eliminate it. We believe that, with the single fsync, the kernel flushed old checkpoint pages only when it had to—when the buffer cache became full—placing undue stress on the rest of the system. Frequent synchronization, (c), produces far more stable results; it also can produce a checkpoint more quickly than can the version with occasional sleeps.

Compression. We also experimented with compressing the database checkpoints via lz4 before writing to disk. This didn't help either latency or throughput, and it actually slowed down the time it took to checkpoint. Our storage is fast enough that the cost of checkpoint compression outweighed the benefits of writing less data.

6.3 On-line transaction processing workload

YCSB-A, though challenging, is a well-behaved workload: all records are in one table, there are no secondary indexes, accesses are uniform, all writes are overwrites (no inserts or deletes), all transactions are small. In this section, we evaluate SiloR on a more complex workload, the popular TPC-C benchmark for online transaction processing [26]. TPC-C transactions involve customers assigned to a set of districts within a local warehouse, placing orders in those districts. There are ten primary tables plus two secondary indexes (SiloR treats primary tables and secondary indexes identically). We do not model client "think" time, and we run the standard workload mix. This contains 45% "new-order" transactions, which contain 8–18 inserts and 5–15 overwrites each. Also write-heavy are "delivery" transactions (4% of the mix), which contain up to 150 overwrites and 10 removes each.[2] Unmodified TPC-C is not a great fit for an in-memory database: very few records are removed, so the database grows without bound. During our 10-minute experiments, database record size (not including keys) grows from 2 GB to 94 GB. Nevertheless, the workload is well understood and challenging for our system.

Figure 5 shows the results. TPC-C transactions are challenging enough for Silo's in-memory structures that the addition of persistence has little effect on throughput: SiloR's throughput is about 93% that of MemSilo. The MemSilo graph also shows that this workload is more inherently variable than YCSB-A. We use 28 workers for MemSilo, rather than 32, because 32-worker runs actually have lower average throughput, as well as far more variability (see Figure 7 in the appendix: our 28-worker runs achieved 587–596 Mtxn/s, our 32-worker runs 565–583 Mtxn/s). As with YCSB-A, the addition of persistence increases this variability, both by batching transactions and by further stressing the machine. (Figure 7 in the appendix shows that, for example, checkpoints can happen at quite different times.) Throughput degrades over time in the same way for all configurations. This is because the database grows over time, and Silo tables are stored in trees with height proportional to the log of the table size. The time to take a check-

[2]It is common in the literature to report TPC-C results for the standard mix as "new order transactions per minute." Following Silo, we report transactions per second for *all* transactions.

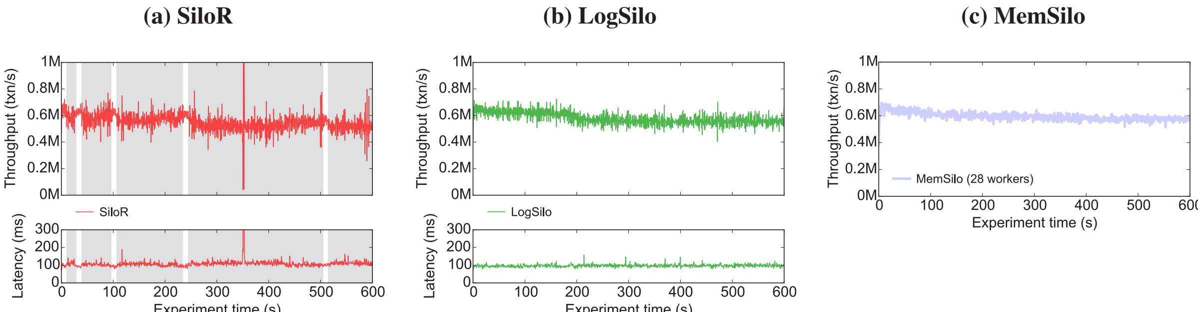


Figure 5: Throughput and latency of SiloR and LogSilo, and throughput of MemSilo, on a modified TPC-C benchmark. Average throughput is 548 Ktxn/s, 575 Ktxn/s, and 592 Ktxn/s, respectively. Average SiloR latency is 110 ms/txn; average LogSilo latency is 97 ms/txn. The database initially contains 2 GB of record data, and grows to 94 GB by the end of the experiment. All experiments run 28 workers.

point also grows with database size (3.5 s or so per GB of record data). Latency, which is 110 ms/txn average for SiloR, is higher than in YCSB-A, but not by much, even though TPC-C transactions are far more complex. In summary, SiloR can handle more complex workloads with larger transactions as well as it can handle simple workloads with small transactions.

6.4 Recovery

We now show that SiloR checkpoints allow for fast recovery. We run YCSB-A and TPC-C benchmarks, and in each case, crash the database immediately before a checkpoint completes. This maximizes the length of the log that must be recovered to restore a transactionally-correct state. We use 6 threads per disk (24 threads total) to restore the checkpoint, and 8 threads per disk (32 threads total) to recover the log.

For YCSB-A, SiloR must recover 36 GB of checkpoint and 64 GB of log to recreate a 43.2 GB database. Recovery takes 106 s, or about 1.06 s/GB of recovery data. 31% of this time (33 s) is spent on the checkpoint and the rest (73 s) on the log. The TPC-C database grows over time, so checkpoints have different sizes. We stop a SiloR run of TPC-C immediately before its fourth checkpoint completes, at about 465 s into the experiment, when the database contains about 72.2 GB of record data (not including keys). SiloR must recover 15.7 GB of checkpoint and 180 GB of log to recreate this database. Recovery takes 211 s, or about 1.08 s/GB of recovery data. 8% of this time (17 s) is spent on the checkpoint and the rest (194 s) on the log. Thus, recovery time is proportional to the amount of data that must be read to recover, and log replay is the limiting factor in recovery, justifying our decision to checkpoint frequently.

6.5 Discussion

This work's motivation was to explore the performance limits afforded by modern hardware. However, there are other limits that SiloR would encounter in a real deployment. At the rates we are writing, our expensive flash drives would reach their maximum endurance in a bit more than a year!

In contrast with the evaluation of Silo, we disable the NUMA-aware allocator in our tests. When enabled, this allocator improves average throughput by around 25% on YCSB (to 10.91 Mtxn/s for SiloR) and 20% on TPC-C (to 644 Ktxn/s for SiloR). The cost—which we decided was not worth paying, at least for TPC-C—was performance instability and dramatically worse latency. Our TPC-C runs saw sustained latencies of over a second in their initial 40 s so, and frequent latency spikes later on, caused by fsync calls and writes that took more than 1 s to complete. These slow file system operations appear unrelated to our storage hardware: they occur only when two or more disks are being written simultaneously; they occur at medium write rates as well as high rates; they occur whether or not our log and checkpoint files are preallocated; and they occur occasionally on each of our disks (both Fusion and RAID). Turning off NUMA-aware allocation greatly reduces the problem, but traces of it remain: the occasional latency spikes visible in our figures have the same cause. NUMA-aware allocation is fragile, particularly in older versions of Linux like ours;[3] it is possible that a newer kernel would mitigate this problem.

7 Related work

SiloR is based on Silo, a very fast in-memory database for multicore machines [27]. We began with the publicly available Silo distribution, but significantly adapted the logging implementation and added checkpointing and recovery. Silo draws from a range of work in databases

[3]For instance, to get good results with the NUMA allocator, we had to pre-fault our memory pools to skirt kernel scalability issues; this step could take up to 30 minutes per run!

and in multicore and transactional memory systems more generally [1, 3, 6–9, 11, 12, 15, 18, 19, 21].

Checkpointing and recovery for in-memory databases has long been an active area of research [5, 20, 22–24]. Salem et al. [24] survey many checkpointing and recovery techniques, covering the range from fuzzy checkpoints (that is, inconsistent partial checkpoints) with value logging to variants of consistent checkpoints with operation logging. In those terms, SiloR combines an action-consistent checkpoint (the transaction might contain some, but not all, of an overlapping transaction's effects) with value logging. Salem et al. report this as a relatively slow combination. However, the details of our logging and checkpointing differ from any of the systems they describe, and in our measurements we found that those details matter. In Salem et al. action-consistent checkpoints either write to all records (to paint them), or copy concurrently modified records; our checkpointers avoid all writes to global data. More fundamentally, we are dealing with database sizes and speeds many orders of magnitude higher, and technology tradeoffs may have changed.

H-Store and its successor, VoltDB, are good representatives of modern fast in-memory databases [10, 13, 25]. Like SiloR, VoltDB achieves durability by a combination of checkpointing and logging [14], but it makes different design choices. First, VoltDB uses command logging (a variant of operation logging), in contrast to SiloR's value logging. Since VoltDB, unlike Silo, partitions data among cores, it can recover command logs somewhat in parallel (different partitions' logs can proceed in parallel). Command logging in turn requires that VoltDB's checkpoints be transactionally consistent; it takes a checkpoint by marking every database record as copy-on-write, an expense we deem unacceptable. Malviya et al. also evaluate a variant of VoltDB that does "physiological logging" (value logging). Although their command logging recovers transactions not much faster than it can execute them—whereas physiological logging can recover transactions 5x faster—during normal execution command logging performs much better than value logging, achieving 1.5x higher throughput on TPC-C. This differs from the results we observed, where value logging was just 10% slower than a system with persistence entirely turned off. Our raw performance results also differ from those of Malviya et al. For command logging on 8 cores, they report roughly 1.3 Ktxn/s/core for new-order transactions, using a variant of TPC-C that entirely lacks cross-warehouse transactions. (Cross-warehouse transactions are particularly expensive in the partitioned VoltDB architecture.) Our TPC-C throughput with value logging, on a mix including cross-warehouse transactions and similar hardware, is roughly 8.8 Ktxn/s/core for new-order transactions. Of course, VoltDB is more full-featured than SiloR.

Cao et al. [2] describe a design for frequent consistent checkpoints in an in-memory database. Their requirements align with ours—fast recovery without slowing normal transaction execution or introducing latency spikes—but for much smaller databases. Like Malviya et al., they use "logical logging" (command/operation logging) to avoid the expense of value logging. The focus of Cao et al.'s work is two clever algorithms for preserving the in-memory state required for a consistent checkpoint. These algorithms, Wait-Free ZigZag and Wait-Free Ping-Pong, effectively preserve 2 copies of the database in memory, a current version and a snapshot version; but they use a bitvector to mark on a per-record basis which version is current. During a checkpoint, updates are directed to the noncurrent version, leaving the snapshot version untouched. This requires enough memory for at least 2, and possibly 3, copies of the database, which for the system's target databases is realistic (they measure a maximum of 1.6 GB). As we also observe, the slowest part of recovery is log replay, so Cao et al. aim to shorten recovery by checkpointing every couple seconds. This is only possible for relatively small databases. Writing as fast as spec sheets promise, it would take at least 10 seconds for us to write a 43 GB checkpoint in parallel to 3 fast disks, and that is assuming there is no concurrent log activity, and thus that normal transaction processing has halted.

The gold standard for database logging and checkpointing is agreed to be ARIES [16], which combines undo and redo logging to recover inconsistent checkpoints. Undo logging is necessary because ARIES might flush uncommitted data to the equivalent of a checkpoint; since SiloR uses OCC, uncommitted data never occurs in a checkpoint, and redo logging suffices.

The fastest recovery times possible can be obtained through hot backups and replication [14, 17]. RAMCloud, in particular, replicates a key-value store node's memory across nearby disks, and can recover more than 64 GB of data to service in just 1 or 2 seconds. However, RAMCloud is not a database: it does not support transactions that involve multiple keys. Furthermore, RAMCloud achieves its fast recovery by fragmenting failed partitions across many machines. This fragmentation is undesirable in a database context because increased partitioning requires more cross-machine coordination to run transactions (e.g., some form of two-phase commit). Nevertheless, 1 or 2 seconds is far faster than SiloR can provide. Replication is orthogonal to our system and an interesting design point we hope to explore in future work.

8 Conclusions

We have presented SiloR, a logging, checkpointing, and recovery subsystem for a very fast in-memory database. What distinguishes SiloR is its focus on performance for extremely challenging workloads. SiloR writes logs and checkpoints at gigabytes-per-second rates without greatly affecting normal transaction throughput, and can recover > 70 GB databases in minutes.

For future work, we would like to investigate checkpointers that cycle through logical partitions of the database. We believe this approach will allow us to substantially reduce the amount of log data that needs to be replayed after a crash. Another possibility is to investigate a RAMCloud-like recovery approach in which data is fragmented during recovery, allowing quick resumption of service at degraded rates, but then reassembled at a single server to recover good performance.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Allen Clement, for helpful comments and patience. This work was supported by the NSF under grants 1302359, 1065219, and 0704424, and by Google and a Microsoft Research New Faculty Fellowship.

References

[1] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. PPoPP '10*, Jan. 2010.

[2] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proc. ACM SIGMOD 2011*, June 2011.

[3] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proc. VLDB '01*, Sept. 2001.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. ACM Symp. on Cloud Computing*, June 2010.

[5] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. SIGMOD '84*, June 1984.

[6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proc. SIGMOD 2013*, June 2013.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. DISC '06*, Sept. 2006.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11), 1976.

[9] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. 12th Conf. on Extending Database Tech.*, Mar. 2009.

[10] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1:1496–1499, 2008.

[11] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2), 1981.

[12] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4), 2011.

[13] A.-P. Liedes and A. Wolski. SIREN: a memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In *Proc. ICDE '06*, Apr. 2006.

[14] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *Proc. ICDE '14*, Mar. 2014.

[15] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys '12*, Apr. 2012.

[16] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. on Database Sys.*, 17(1):94–162, 1992.

[17] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proc. SOSP 2011*, Oct. 2011.

[18] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2), 2010.

[19] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page latch-free shared-everything OLTP. *Proc. VLDB Endow.*, 4(10), 2011.

[20] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1:271–287, 1986.

[21] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *Proc. VLDB Endow.*, 6(2), 2012.

[22] D. Rosenkrantz. Dynamic database dumping. In *Proc. SIGMOD '78*, May 1978.

Appendix

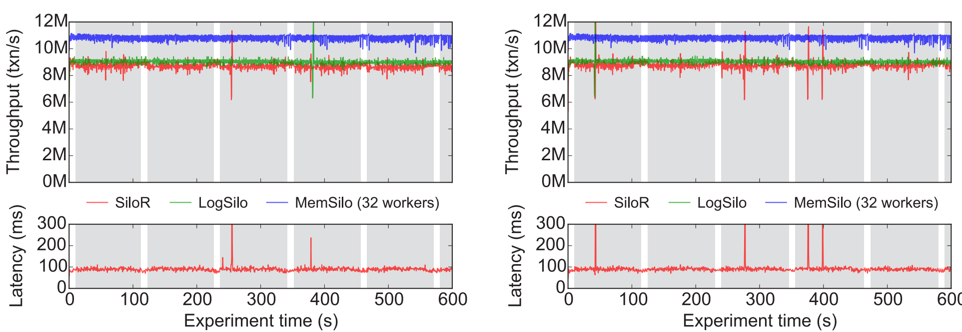


Figure 6: Performance of SiloR, LogSilo, and MemSilo on our modified YCSB benchmark: additional runs.

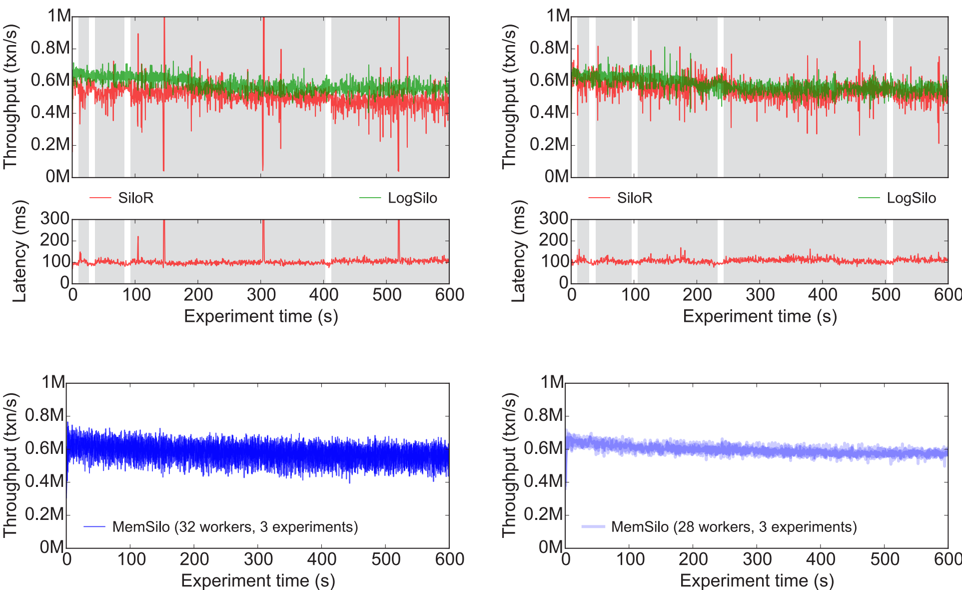


Figure 7: Performance of SiloR, LogSilo, and MemSilo (with 32 and 28 workers) on our modified TPC-C benchmark: additional runs.

[23] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *Proc. ICDE '89*, Feb. 1989.

[24] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Trans. Knowledge and Data Eng.*, 2(1), Mar. 1990.

[25] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. VLDB '07*, Sept. 2007.

[26] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, June 2007.

[27] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. SOSP '13*, Nov. 2013.

Extracting More Concurrency from Distributed Transactions

Shuai Mu[†‡], Yang Cui[‡], Yang Zhang[‡], Wyatt Lloyd[♯♮], Jinyang Li[‡]
[†]*Tsinghua University,*[*] [‡]*New York University,* [♯]*University of Southern California,* [♮]*Facebook*

Abstract

Distributed storage systems run transactions across machines to ensure serializability. Traditional protocols for distributed transactions are based on two-phase locking (2PL) or optimistic concurrency control (OCC). 2PL serializes transactions as soon as they conflict and OCC resorts to aborts, leaving many opportunities for concurrency on the table. This paper presents ROCOCO, a novel concurrency control protocol for distributed transactions that outperforms 2PL and OCC by allowing more concurrency. ROCOCO executes a transaction as a collection of atomic pieces, each of which commonly involves only a single server. Servers first track dependencies between concurrent transactions without actually executing them. At commit time, a transaction's dependency information is sent to all servers so they can re-order conflicting pieces and execute them in a serializable order.

We compare ROCOCO to OCC and 2PL using a scaled TPC-C benchmark. ROCOCO outperforms 2PL and OCC in workloads with varying degrees of contention. When the contention is high, ROCOCO's throughput is 130% and 347% higher than that of 2PL and OCC.

1 Introduction

Many large-scale Web services, such as Amazon, rely on a distributed online transaction processing (OLTP) system as their storage backend. OLTP systems require concurrency control to guarantee strict serializability [12, 13], so that websites running on top of them can function correctly. Without strong concurrency control, sites could sell items that are out of stock, deliver items to customers twice, double-charge a customer for a sale, or indicate to a customer they did not purchase an item they actually did.

While concurrency control is a well-studied field, traditional protocols such as two-phase locking (2PL) [12] and optimistic concurrency control (OCC) [36] perform poorly when workloads exhibit a non-trivial amount of contention [8, 30]. The performance drop is particularly pronounced when running these protocols in a distributed setting. When there are many conflicting concurrent transactions, 2PL and OCC abort and retry many of them, leading to low throughput and high latency. In our evaluation in § 5, the throughput of 2PL and OCC drops to less than 10% of its maximum as contention increases.

Unfortunately, contention is not rare in large-scale OLTP applications. For example, consider a transaction where a customer purchases a few items from a shopping website. Concurrent purchases by different customers on the same item create conflicts. Moreover, as the system scales—i.e., the site becomes more popular and has more customers, but maintains a relatively stable set of items—concurrent purchases to the same item are more likely to happen, leading to a greater contention rate.

In this paper we presents ROCOCO (ReOrdering COnflicts for COncurrency), a distributed concurrency control protocol that extracts more concurrency under contended workload than previous approaches. ROCOCO achieves safe interleavings without aborting or blocking transactions using two key techniques: 1) deferred and reordered execution using dependency tracking [38, 46]; and 2) offline safety checking based on the theory of transaction chopping [50, 49, 57].

ROCOCO is a two round protocol that executes transactions that have been structured into a collection of atomic pieces, each typically involving data access on a single server. A set of coordinators run the protocol on behalf of clients. The first phase distributes the pieces to the appropriate servers and establishes a provisional order of execution on each server. Servers typically defer execution of the pieces until the second round so they can be reordered if necessary. Servers complete the first phase by replying to the coordinator with dependency information that indicates the order of arrival for conflicting pieces of different transactions.

The coordinator aggregates this dependency information and distributes it to all involved servers. Servers use the aggregated dependency information to recognize if the pieces of concurrent transactions arrived at servers in a strictly serializable order in the first phase. If so, they execute pieces in that order in the second phase. If not, servers reorder the pieces deterministically and then exe-

[*]The full name is Tsinghua National Laboratory for Information Science and Technology (TNLIST), Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

cute them. In both cases, ROCOCO is able to avoid aborts and commits all transactions.

Dependencies are usually exchanged only between servers and coordinators in the two round protocol. But, when conflicting transactions have overlapping but non-identical sets of servers, ROCOCO occasionally requires additional server-to-server communication to ensure a deterministic order.

Not all transaction pieces can have their execution deferred to the second round, e.g., a piece that reads a value to determine what data item to access next. Such pieces must be executed immediately in ROCOCO's first phase, which can result in un-serializable interleavings. To ensure that a strictly serializable reordering is always possible during runtime, ROCOCO performs an offline check on the transaction workload prior to starting the transactions. The offline checker identifies and categorizes potential conflicts. If some pieces of a transaction are found to have unsafe interleaving that cannot be reordered, ROCOCO merges those pieces into a single atomic piece. While a traditional concurrency control protocol is used to execute a merged piece across servers atomically, the ROCOCO protocol is used to execute multiple merged pieces within a transaction.

We implemented ROCOCO and evaluated its performance using a scaled TPC-C benchmark [5]. ROCOCO supports the TPC-C workload without requiring any merged pieces and avoids ever aborting. ROCOCO outperforms 2PL and OCC in workloads with varying degrees of contention. When the contention is high, ROCOCO's throughput is 130% and 347% higher than that of 2PL and OCC. As the system scales across TPC-C warehouse districts and contention increases, the throughput of ROCOCO continues to grow while the throughput of OCC drops to almost zero and 2PL does not scale.

2 Overview

ROCOCO targets OLTP workloads in large-scale distributed database systems, e.g., the backend of e-commerce sites like Amazon. For scalability, database tables are sharded row-wise across multiple servers, with each server holding a subset of certain tables. Thus, a transaction accessing different table rows typically needs to contact more than one server and requires a distributed concurrency control protocol.

For performance, we assume a setup where transactions are executed as *stored procedures*, as in earlier work [34, 52, 26, 28, 41, 56, 55]. Specifically, a distributed transaction consists of a set of stored procedures called *pieces*. Each piece accesses one or more data items stored on a single server using user-defined logic. Thus, each piece can be executed atomically with respect to other concurrent pieces by employing proper local concurrency control. We assume stored procedures are distributed to all servers apriori because they have a minimal storage costs.

```
transaction new_order_fragment:
  #simplified new-order "buys" 1 of item_a,item_b
  input: item_a, item_b
  begin
        ...
  p1:   # reduce stock level of item_a
        R(tab="Stock", key=item_a) → stock
        if (stock > 1):
          W(tab="Stock", key=item_a) ← stock - 1
        ...
  p2:   # reduce stock level of item_b
        R(tab="Stock", key=item_b) → stock
        if (stock > 1):
          W(tab="Stock", key=item_b) ← stock - 1
        ...
  end
```

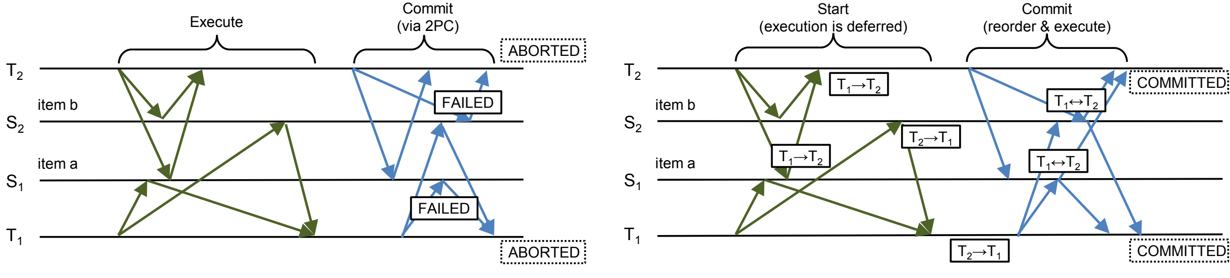
Figure 1: A fragment of TPC-C new-order transaction containing two pieces.

2.1 Traditional Approaches Abort Conflicts

Application programmers prefer the strongest isolation level, strict serializability [12, 31], to simplify the reasoning of correctness in the face of concurrent transactions. To guarantee strict serializability, a distributed storage system typically runs standard concurrency control schemes such as two-phase locking (2PL) or optimistic concurrency control (OCC), combined with two-phase commit (2PC) [19].

2PL and OCC perform poorly for contended workload with many conflicting transactions. As an example, consider a simplified fragment of the TPC-C new-order transaction which simulates a customer purchasing two items from a store (Figure 1). The transaction contains two stored procedure pieces, p_1 and p_2, each of which reduces the stock level of a different item. Although each piece can be executed atomically on its server, distributed concurrency control is required to prevent non-serializable interleaving of pieces across servers. For instance, suppose a merchant keeps the same stock level for item a (e.g., a xbox) and item b (e.g., a xbox controller) and always sells the two items as bundles. Without distributed concurrency control, one customer could receive an item a, but not item b, while another customer could receive item b but not item a.

We first examine the behavior of OCC with two transactions, T_1 and T_2. Both purchase the same two items, a and b, that are stored on different servers. Any interleaving of T_1 and T_2's pieces during execution causes aborts when performing OCC validation during 2PC. For example, if T_2 reads the stock level of a after T_1 reads it, but



(a) OCC **(b) ROCOCO**

Figure 2: A possible interleaving of two concurrent new-order transaction fragments. In the left figure (OCC), both transactions fail to validate and abort. In the right figure (ROCOCO), the interference is captured by dependencies and the transactions are reordered to a strictly serializable order before execution.

before T_1 commits its update to a, then T_2 will later fail to validate and abort. Figure 2a shows another example where both T_1 and T_2 are aborted during 2PC because their corresponding 2PC prepare messages are handled by servers in different orders.

2PL outperforms OCC under contention but is still far from satisfactory. 2PL acquires locks for each data access, which serializes the execution of transactions as soon as they perform a conflicting operation. In the new-order example, as soon as T_1 modifies the stock level of a, T_2 is blocked until T_1 completes all of its pieces and commits. In addition to blocking, 2PL also resorts to aborts to prevent deadlocks [19]. As the amount of contention increases, so does the probability of having a deadlock. Furthermore, efficient deadlock prevention mechanisms such as wound-wait [48] have many false positives, thereby causing a large number of aborts even when there is no real deadlock.

2.2 ROCOCO Reorders Conflicts to Commit

Given conditions discussed later, our new concurrency control protocol, ROCOCO, avoids aborting or blocking under contention by identifying and then avoiding interference between transactions. Two transactions *interfere* when executing their constituent pieces in their arrival order at each server would result in a non-serializable execution. For example, T_1 and T_2 interfere in Figure 2b because their pieces arrive in different orders on servers S_1 and S_2. If both pieces of T_1 arrived before both pieces of T_2 they would not interfere.

ROCOCO tracks potential interference using dependency information between pieces of transactions that are generated when pieces *conflict* on a server, i.e., both access the same data location and at least one of them writes to it. Servers use dependency information to detect if transactions interfere and deterministically reorder their pieces so they are executed in the same order on all involved servers and, thus, no longer interfere.

ROCOCO is able to change the order of execution of pieces because it uses two rounds of messages to commit them. The first round starts with a transaction coordinator running on behalf of a client disseminating the pieces of a transaction to the appropriate servers. The servers do not yet execute the pieces and instead return dependency information to the coordinator to complete the first round. The coordinator then combines all the dependency information and distributes it to all the servers in the second round. The servers then reorder pieces of the transaction, if necessary, before executing them.

Figure 2b shows an example of ROCOCO in action. S_1 observes $T_1 \rightarrow T_2$, reflecting the arrival order of the conflicting pieces it has received from T_1 and T_2. Similarly, S_2 observes $T_2 \rightarrow T_1$. The coordinator collects $T_1 \leftrightarrow T_2$ and sends this dependency information to both servers. The servers recognize the cycle of interference and deterministically order the involved transactions and thus their constituent pieces before executing them. The ordering of the two transactions can be any deterministic order, e.g., the order of their globally unique transactions ids, which in the example would execute T_1 and then T_2. With ROCOCO, T_1 and T_2 both commit and neither has to abort or wait for the other.

By reordering interfering transactions instead of aborting them, ROCOCO can achieve significant performance improvement when there is a non-trivial amount of contention, which is often the case with OLTP workloads. For example, a complete TPC-C new order transaction updates a highly contended *order-id* data field as well as 10 purchased items on average. As the number of concurrent requests rises, the probability of contending on a purchased item also increases. Moreover, the power-law distribution often seen in real-world workloads results in even higher contention on "hot" items.

3 Design

The design of ROCOCO includes an offline checker and a runtime protocol. The offline checker determines if the pieces of a collection of transactions can be reordered correctly at runtime. The runtime protocol tracks the dependencies between pieces and reorders their execution if necessary for correctness.

In this section, we explain ROCOCO's offline check (§ 3.1), runtime protocol (§ 3.2), and sketch its correctness (§ 3.3). We then discuss an important optimization (§ 3.4) and the fault tolerance mechanism (§ 3.5).

3.1 Checking When Reordering is Viable

Reordering the execution of pieces of interfering transaction is only possible under certain, common, conditions. This subsection explains the difference between immediate pieces of a transaction that cannot be reordered and deferrable pieces that can. Then it explains how ROCOCO's offline checker uses transaction profiles including immediate/deferrable information to check for the necessary conditions.

Immediate and deferrable pieces. A piece of a transaction is either immediate or deferrable depending on the stored procedure it executes. If the output of a piece p can serve as the input to another piece p', then p is an *immediate* piece because it must be executed before its parent transaction can move on to its subsequent pieces. Conversely, a piece is *deferrable* if its output is not required by any other piece. A server can postpone the execution of a deferrable piece until the commit time of a transaction.

Once executed, immediate pieces cannot be reordered, which can result in a non-serializable interleaving. As an example, suppose p_1 and p_2 in Figure 1 are both immediate instead of deferrable pieces. Then Figure 2's message interleaving makes a total ordering of the transactions impossible. In particular, S_1 executes piece p_1 of T_1 before that of T_2, fixing $T_1 \rightarrow T_2$ in the total order. However, the execution on S_2 fixes $T_2 \rightarrow T_1$ in the total order, a contradiction.

If at least one of the pieces is deferrable, however, a total order can be achieved. For example, instead suppose p_1 is immediate and p_2 is deferrable, then the interleaving can be reordered at S_2 so that p_2 of T_1 is executed before that of T_2, i.e., $T_1 \rightarrow T_2$, which is consistent with the execution at S_1 and thus a total order. ROCOCO's offline checker ensures that there exists such a deferrable piece for all sets of possibly interfering transactions.

The offline checker. In order to ensure that a serializable reordering of conflicting pieces is always possible at runtime, ROCOCO relies on an offline checker that analyzes the conflict profile of *all* transactions to be executed. Fortunately, OLTP workloads typically have a fixed set of transactions that are known apriori [52], making such an offline checker practical.

To build ROCOCO's offline checker, we extend the theory of transaction chopping [50, 57]. For each piece p, we assume the checker knows whether p is an immediate or deferrable piece and the database tables and columns p reads or writes. We do not assume the checker knows which rows p accesses. In our current implementation, programmers explicitly write each transaction as a set of pieces and manually annotate each piece's type and database accesses.

The checker works in several steps. First, it constructs a SC-graph, similar to earlier uses of transaction chopping [50, 57]. Each transaction appears as two instances in the graph where each piece is a vertex and pieces from the same transaction instance are connected by *S(ibling)-edges*. If two pieces access the same database table and at least one of the accesses is a write, they are connected by a *C(onflict)-edge*. If a cycle in the graph contains both S- and C-edges, it is a *SC-cycle*. Each SC-cycle signals a potential conflict that can lead to non-serializable execution [50, 57].

Next, the checker tags each vertex as either an *I(mmediate)* or *D(eferrable)* piece. The checker virally propagates immediacy across C-edge by changing the tag of any piece with a C-edge to an I piece to also be I until there are no C-edges between pieces with different I/D tags. We refer to a C-edge as I-I (or D-D) if both end points are I (or D) pieces. There are no I-D edges.

Finally, the checker examines if there exists an *unreorderable* SC-cycle where all C-edges are I-I edges. If there are none, ROCOCO's basic protocol can safely reorder all conflicts to ensure serializability at runtime. Intuitively, SC-cycles represent potential non-serializable interleavings [49]. However, if an SC-cycle contains at least one D-D edge, ROCOCO can reorder the execution of the D-D edge's pieces to break the cycle, thereby ensuring serializability. For an unreorderable SC-cycle with all I-I C-edges, the checker proposes to merge those pieces in the cycle belonging to the same transaction into a larger atomic piece. In the later section § 4.2, we explain how ROCOCO relies on traditional distributed concurrency control methods such as 2PL or OCC to execute merged pieces.

Figure 3 shows a more complete version of the TPC-C new-order transaction that includes two new pieces in addition to the stock-level-reduction piece discussed earlier. p_1 reads the next order id (`next_oid`), increments it, and writes it back. p_2 modifies the stock level of the purchased item. There may be many instances of p_2, depending on how many items the customer buys, denoted p_2', p_2'', etc. p_3 records the order information in the

```
transaction simplified_new_order:
  input: [item_a, item_b, ...], district d
  begin
        ...
p1: #pick the next order id
   R(tab="District",col="next_oid",key=d)→ oid
   W(tab="District",col="next_oid",key=d)← oid+1

p2: #reduce the stock level of each item
    #(one piece for each item)
   R(tab="Stock", key=item)→ stock
   if (stock > 1):
     W(tab="Stock", key=item)← stock-1
   ...

p3: #add orderline info for each item
    #(one piece for each item)
   W(tab="OrderLine", key=item+oid) ← ...
   ...
  end
```

Figure 3: A simplified TPC-C new-order transaction

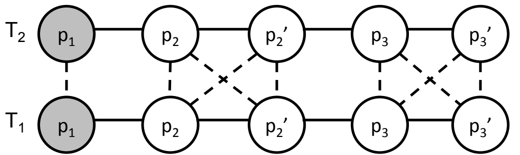


Figure 4: SC-graph of the TPC-C new-order sample transaction. Gray circles represent immediate pieces; white circles represent deferrable pieces. Solid lines represent S-edges; dotted lines represent C-edges. ROCOCO can safely execute this transaction workload because all SC-cycles include at least one D-D edge.

database using the order id output by p_1. There may also be multiple instances of p_3, denoted p_3', p_3'', etc. p_1 is an immediate piece because p_3 reads from it while p_2 and p_3 are deferrable pieces. Figure 4 shows the SC-graph of a workload that only contains concurrent new-order transactions that buy two items. ROCOCO can safely execute this workload because all SC-cycles in the graph have a D-D edge.

User-initiated aborts. Previous systems based on transaction chopping [50, 57] sequentially execute pieces and allow user-initiated aborts only in the first piece. ROCOCO, in contrast, executes pieces in parallel so there is no natural "first" piece. For simplicity, we disallow all user-initiated aborts. [1]

3.2 Basic Protocol

ROCOCO's runtime protocol executes a collection of transactions deemed safe for reordering by the offline checker. Clients delegates the responsibility of coordinating their transactions to separate *coordinator* processes. There can be many coordinators and a typical deployment co-locates coordinators with servers.

Once a coordinator receives a client's transaction request, it processes the transaction in two phases: start and commit. In the *start phase*, the coordinator sends pieces to servers and collects the returned dependency information. In the *commit phase*, the coordinate disseminates the aggregated dependency information to all participating servers who reach a deterministic serializable order to execute conflicting transactions.

Figure 5 shows a typical message flow for ROCOCO.

The start phase. The start phase of a transaction distributes its pieces, sets a provisional order for them on servers, executes immediate pieces, and collects dependency information.

The start phase begins when the coordinator sends out requests for all pieces of a transaction together with their inputs to the appropriate servers—i.e., the servers that store the items read or written by those pieces. If a piece p is immediate, the server will execute p immediately and return its output so that the coordinator can proceed to issue other pieces whose inputs are based on p's output. If p is deferrable, the server buffers it for later execution. The coordinator also parallelizes the issuing of requests when possible, only blocking a request if its inputs are not yet available.

In addition to executing immediate pieces and buffering deferrable ones, each server maintains a dependency graph, *dep*. Each vertex in *dep* represents a transaction and its known status, which can be any one in the ordered set {STARTED, COMMITTING, DECIDED}. In addition, for each transaction T involving server S, S keeps a boolean flag $T.finished$ to indicate whether server S has finished committing T. Each edge represents the order of conflicting pieces between two transactions as observed by the server. For example, if a server receives p_1 that writes to data item x. Then, upon receiving p_2 that also accesses x, the server adds a direct edge $p_1.owner \rightarrow p_2.owner$ to *dep*, where $p.owner$ denotes p's corresponding transaction. Moreover, each edge is labeled depending on the types of p_1 and p_2 as immediate or deferred. If both pieces are immediate, the edge is labeled as $\xrightarrow{i}$; if both are deferrable, the edge is $\xrightarrow{d}$. There cannot be an edge between an immediate and deferrable piece because the offline checker eliminated such scenarios when it virally propagated immediacy over C-edges.

Figure 10 summarizes how a server processes a start request in pseudocode. The server returns its updated *dep* graph and the piece's execution output if the piece is immediate to the coordinator. The coordinator sim-

[1] User-initiated aborts are important if the transaction needs to terminate after it has written to the database, which means all writes need rollback. If the aborts happen before any writes, it can be replaced with simple termination.

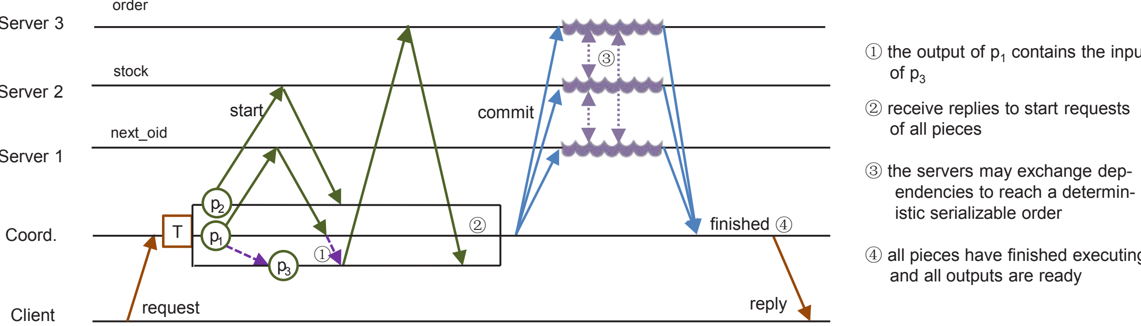


Figure 5: A typical ROCOCO message flow.

```
function Server S::start_req(p):
  S.dep[p.owner].status = STARTED
  foreach p' received by S that conflicts with p
    if p.immediate == true
      add p'.owner -i-> p.owner to S.dep
    else
      add p'.owner -d-> p.owner to S.dep
  if p.immediate == true
    output = execute(p)
  return (S.dep, output)
```

Figure 6: How a server processes a start request.

ply aggregates the returned dependency graphs from all involved servers (not shown in psuedocode).

The commit phase. The commit phase of a transaction distributes dependency information for all pieces, ensures each server can safely decide if a piece must be reordered, deterministically reorders pieces on each server if necessary, executes deferred pieces, and commits a transaction.

The coordinator begins the commit phase once it has sent out all start requests and collected their responses. For each participating server, the coordinator sends a commit request containing the aggregated *dep* graph. When aggregating a set of dependency graphs, one takes the union of vertexes/edges and sets each vertex's status to be the highest one in those graphs.

Figure 7 summarizes how a server handles a commit request in psuedocode. Upon receiving a commit request for transaction T, server S updates the status of T to COMMITTING in its dependency graph, if $T.status$ is lower than COMMITTING. Server S also aggregates the dependency information in the commit request into $S.dep$.

Next, server S ensures it can safely decide if its piece of T should be reordered by collecting the transitive closure of T's conflicting transactions' in $S.dep$. To do this, it examines $S.dep$ to find all T' that are ancestors of T and waits for the status of those T' to become COMMITTING or DECIDED. In the common case when T' involves server S and S will eventually receive the commit request of T' so it simply waits; in the uncommon case when T' does not involve S, it issues a status request for T' to a server S' involved in T'. S' replies with its dependency graph after S' has received the commit request of T'. S aggregates the received graph with its own.

Next, server S calculates the strongly connected component (SCC) of T in $S.dep$, denoted T^{SCC}, which typically includes only T. [2] The server then sets the status of all transactions in T^{SCC} to DECIDED. Next, the server waits for all ancestors of the T^{SCC} to become DECIDED. Furthermore, for each ancestor T' involving server S, S also waits for $T'.finished$ to become true.

Next, to decide the right execution order for T, server S topologically sorts T^{SCC} according to $\xrightarrow{i}$ edges. To ensure that different servers reach a single sorting order, sorting is done deterministically. This topological sort is possible if and only if there are no cycles in T^{SCC} connected by only $\xrightarrow{i}$ edges. ROCOCO's offline checker ensures this will always be the case by eliminating any SC-cycle whose C-edges all have the I-I type. We elaborate this argument further in § 3.3.

Finally, server S executes the deferred pieces of each transaction T in T^{SCC} that involves S in the sorted order. Upon finishing executing T, server S sets $T.finished$ to be true and returns the results to T's coordinator.

When a coordinator has collected the responses from all participating servers, the transaction is considered committed and the output is returned to the client.

3.3 Correctness

This subsection presents a proof sketch of correctness. A more rigorous version of the proof is available in a technical report [47]. Specifically, we prove that ROCOCO

[2] We use the Tarjan algorithm [53] for SCC computation. In the best case, only those nodes and edges in the SCC need to be visited; In the worst case, all nodes and edges in the graph need to be visited and the complexity is $O(V+E)$.

```
function Server S::commit_req(T, dep):
  S.dep ∪= dep
  S.dep[T].status ∪= COMMITTING
  foreach T' ⇝ T in dep
    if T' does not involve S and
        S.dep[T'].status == STARTED
      contact S' involved in T' and
        wait until S.dep[T'].status ≥ COMMITTING
  T^SCC = find_SCC(T, S.dep)
  foreach T' not in T^SCC and T' ⇝ T^SCC
    wait until S.dep[T'].status == DECIDED
    if T' involves S
      wait until T'.finished == true
  deterministic_topological_sort(T^SCC)
  foreach T' in T^SCC  # including T
    S.dep[T'].status = DECIDED
    if T' involves S and
        T'.finished == false
      foreach deferred p' of T'
        p'.output = execute(p')
        T'.output ∪= p'.output
        T'.finished = true
  return T.output
```

Figure 7: How a server processes a commit request.

guarantees strict serializability:

Serializability: [12] The committed transactions have an equivalent serial schedule, such that all conflicting operations in the actual schedule are ordered in the same way as in the equivalent serial schedule.

Strict-serializability: [12, 31] The above serial schedule preserves the real-time order, i.e., if transactions T_1 commits before T_2 starts in real time, T_1 appears before T_2 in the equivalent serial schedule.

The proof involves arguments on the *serialization graph*, which is a directed graph where each vertex represents a transaction and each edge represents an ordered conflict. Suppose transactions T_1 and T_2 have conflicting accesses (at least one is a write) to the same data item x. If T_1 accesses x before T_2 does, the serialization graph contains a $T_1 \rightarrow T_2$. To prove ROCOCO is serializable, we must show that any serialization graph it generates is acyclic [13].

First, we show that all relations in the serialization graph are captured in the dependency information collected by servers.

Lemma 1. For any transactions T_1 and T_2, if $T_1 \rightarrow T_2$ is in the serialization graph, then $T_1 \rightarrow T_2$ must be included in the commit request of T_2.

Proof Sketch. By definition, $T_1 \rightarrow T_2$ in the serialization graph implies that a pair of conflicting pieces, p_1 of T_1 and p_2 of T_2, exist and that p_1 executes before p_2 on a corresponding server S. Because the offline checker has eliminated all I-D conflicts, p_1 and p_2 are either 1) both immediate pieces, or 2) both deferrable pieces. In scenario 1), $T_1 \rightarrow T_2$ in the serialization graph means p_1 executes before p_2 in the start phase and server S adds $T_1 \rightarrow T_2$ to $S.dep$. By the ROCOCO protocol, this dependency will be sent back to the coordinator, aggregated with other dependencies, and then appear in T_2's commit requests. In scenario 2), if p_1 executes before p_2, then T_1 has arrived before T_2 at some server S' in the start phase, resulting in $T_1 \rightarrow T_2$ in $S'.dep$. Again, by the ROCOCO protocol, this dependency will be included in T_2's commit request.

Next, we argue that ROCOCO never generates a cycle in the serialization graph, due to a combination of servers breaking SCCs with deferred execution and the offline checker eliminating unreorderable SC-cycles.

Proposition 1. The serialization graph is acyclic.
Proof Sketch. For proof by contradiction, we assume there exists such a cycle (δ) of transactions in the serialization graph. First, we observe that each server involved in δ has δ in its dependency graph prior to executing any transaction in δ in the commit phase. The proof for this observation is in [47] and is based on Lemma 1 and the specification of ROCOCO that ensures each involved server transitively capture conflicting transactions in one SCC. Next, we note that the cycle δ must contain at least one pair of deferrable pieces. If all the pieces in δ are immediate, then δ corresponds to a SC-cycle involving only I-pieces, which would have been detected and eliminated by the offline checker. Last, we obtain a contradiction from the specification of ROCOCO that would have reordered the deferrable pieces to break δ.

Proposition 2. For any transactions T_1 and T_2, if T_2 starts after T_1 has finished, the serialization graph does not contain a path from T_2 to T_1, $T_2 \rightsquigarrow T_1$.
Proof Sketch. To prove by contraction, we assume $T_2 \rightsquigarrow T_1$ exists in the serialization graph. For any $T_i \rightarrow T_j$ in the serialization graph, in order for T_j to become COMMITTING on any server S, ROCOCO requires S to have waited for T_i to become COMMITTING. Therefore, given a path $T_2 \rightarrow T_i \rightarrow T_j \rightarrow .. \rightarrow T_1$ in the serialization graph, we can follow the path in reverse and deduce that T_2 has a status of COMMITTING at some server before T_1 becomes COMMITTING. This implies that T_2 has begun its commit phase before T_1 has finished at all servers, which contradicts the fact that T_2 has not started.

Proposition 1 implies serializability. Proposition 2 additionally shows strict-serializability.

3.4 Reducing Dependency Graphs

In the basic protocol, a server's dependency graph is verbose and grows without bound over time. We now explain how to more efficiently store and transmit dependency information.

To reduce the number of edges in $S.dep$, server S only adds *the nearest dependencies* of T in the graph upon

receiving T's start phase. The nearest dependencies of T have the longest path of one hop to T. However, in contrast to previous work that also tracked the nearest dependencies [42, 43], ROCOCO has two types of edges and paths. If a path contains at least one $\xrightarrow{i}$ edge, it is called an i-path. If a path consists of only $\xrightarrow{d}$ edges, it is called a d-path. An i-path is a stronger type than d-path. A path is longest only if there are no other longer paths with the same or a stronger type.

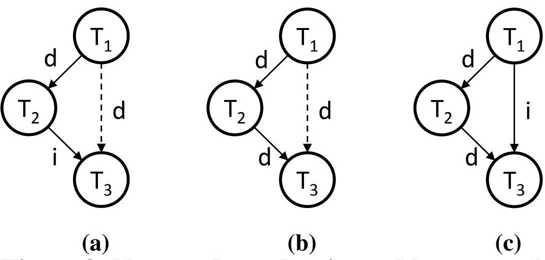


Figure 8: Nearest dependencies and longest paths are shown with solid arrows. $T_1 \xrightarrow{d} T_3$ is not a longest path in the left and middle figures and thus is safely removable. $T_1 \xrightarrow{i} T_3$ is a longest path in the right figure and cannot be omitted.

For example, suppose that server S has $T_1 \xrightarrow{d} T_2$ in $S.dep$. When S receives a deferrable piece of T_3 that conflicts with both T_1 and T_2, instead of adding $\{T_1 \xrightarrow{d} T_3, T_2 \xrightarrow{d} T_3\}$ to $S.dep$, S only adds $\{T_2 \xrightarrow{d} T_3\}$. Skipping $T_1 \xrightarrow{d} T_3$ is acceptable because the dependency is still tracked by $T_1 \xrightarrow{d} T_2$ and $T_2 \xrightarrow{d} T_3$ (Figure 8a). As another example, suppose that $S.dep$ contains $\{T_1 \xrightarrow{d} T_2\}$ and a new transaction T_3 attempts to add $\{T_1 \xrightarrow{i} T_3, T_2 \xrightarrow{d} T_3\}$. In this case, edge $T_1 \xrightarrow{i} T_3$ cannot be skipped, because the path $T_1 \xrightarrow{d} T_2 \xrightarrow{d} T_3$ does not capture the stronger ordering constraint of $T_1 \xrightarrow{i} T_3$ (Figure 8c).

In practice, ROCOCO tracks *one-hop dependencies*, a slightly larger superset of nearest dependencies. When server S receives a new piece p, it finds only the most recent conflicting piece p' for each of p's conflicts and adds $T' \rightarrow T$ to dep. Therefore, if the number of items a piece accesses is constant, then the time and space complexity of handling a new piece is $O(1)$.

In the basic protocol, server S returns the full graph $S.dep$ to the coordinator in the start phase. This is unnecessary. In particular, the coordinator only needs to learn of T's ancestors that are not yet DECIDED. Therefore, server S only computes the subgraph of $S.dep$ containing T's ancestors whose status is lower than DECIDED. Also, in its reply to a status request for T, a server only needs to include every undecided ancestors of T if T is not yet decided; if T is already DECIDED, the server replies with T^{SCC}.

A transaction is considered committed if the coordinator has received commit replies from all involved servers. It is tempting to simply remove all committed transactions from a server's dep. However, it is not correct to do so because the server may receive a status request for its committed transaction from another server. To garbage collect, ROCOCO uses an epoch mechanism similar to previous work [55, 32]. Each server keeps an epoch number that slowly increases. A transaction is tagged with an epoch number when it starts at a server. The epoch number on a server increases only after all transactions in the last epoch are all committed, and no other server falls behind or has ongoing transactions at one or more epochs ago. Dependencies from two epochs ago can be safely discarded.

3.5 Fault Tolerance

To tolerate failure, each server and the coordinator need to persist its transaction log to disks and preferably also replicates it across machines using a Paxos-based replication system [39, 15]. In ROCOCO, the coordinator logs the transaction request before starting the transaction, in case it fails during execution. Each server logs each start request following its arriving order, including its type and input. It does not need to log its output, because the output is deterministic once the order of start requests is fixed.

If a coordinator fails, after it recovers it will send the start requests again to all involved servers. For a server receiving the request, it first examines whether it has received this start request before. If so, it returns the same execution result and dependency graph; If not, it handles this request normally.

If a server fails, when it recovers it needs to replay all the start requests before it responds to other requests. In order to commit these transactions during recovery, the server asks other servers about the corresponding commit requests. In corner cases, such as all servers crashing, the servers should let the coordinator restart the affected transactions. To accelerate the recovery process, the server can also log when a transaction commits (i.e. its finished flag becomes true), but this is off the critical path of a transaction.

4 Extension

We describe two extensions to the basic design of § 3. § 4.1 shows how to optimize read-only transactions. § 4.2 explains how ROCOCO copes with merged pieces that internally rely on traditional distributed concurrency control.

```
function Coordinator C::do_ro_txn(T):
  # chop the transaction into pieces.
  # for a piece p_i, input_i is its input,
  # s_i is its server
  foreach p_i in T
    wait until input_i is ready
    output_i = s_i.ro_req(T, p_i, input_i)
  repeat
    # save the result of the last round read,
    # and issue another round.
    output' = output
    reset(input)
    foreach p_i in T
      wait until input_i is ready
      output_i = s_i.ro_req(T, p_i, input'_i)
    # succeed if the two rounds return the same
  until output = output'
  return output
```

Figure 9: Coordinator read-only transaction

4.1 Read-Only Transactions

Read-only transactions often make up a significant fraction of OLTP workloads. Moreover, they often contain many immediate reads that increase the likelihood of SC-cycles without a D-D edge. To avoid this increase we provide a separate solution to execute read-only transactions that allows the offline checker to exclude read-only transactions from the constructed SC-graph.

To process a read-only transaction, the coordinator sends a round of read requests to each involved server. When a server receives the request it waits for all conflicting transactions to become FINISHED and then it executes the read and returns the result. After the coordinator finishes this round, it issues a second round of requests which are identical to the first round, i.e., they also wait for all conflicting transactions to become FINISHED. The transaction is considered successful if both rounds return the same results. If the results do not match, the coordinator simply re-starts the transaction.

Waiting for all conflicting transaction to finish is the key to ensuring the combination of this read-only transaction algorithm with the rest of ROCOCO is still strictly serializable. When one server waits for a transaction to finish it is forcing that transaction to at least start on all other involved servers. Then, if a first round read happened on a different server before that transaction, its corresponding second round read will at least encounter the transaction in the start phase. The second round read will wait for it to finish before executing, which ensures it will see a different result from the first round and force another round of reads.

4.2 Merged Pieces

In § 3, we assume that the offline checker finds only reorderable SC-cycles so that each piece only involves one

```
function Server S::ro_req(T, p, input):
  foreach T' in S.dep and T' involves S
      and T' conflicts with piece p
    wait until T'.finished is true
  output = execute(p)
  return output
```

Figure 10: Server read-only transaction

server at runtime. When the offline checker discovers unreorderable SC-cycles, it combines the pieces in the cycle that belong to the same transaction into a single *merged piece*. In contrast with the simple pieces discussed above that execute on a single server, a merged piece can be distributed across multiple servers. ROCOCO relies on traditional distributed concurrency control to execute each merged piece atomically across servers.

Fortunately, merged pieces are simple to integrate into the overall design of ROCOCO. A merged piece contains only immediate simple pieces, otherwise, it would not have contributed to an unreorderable SC-cycle. This allows the coordinator to use an OCC-based protocol to execute the sub-pieces of a merged piece in the start phase. Each server returns its dependency information in the normal way.

For example, suppose piece p_2 in Figure 4 is an immediate piece. As a result, p_1 and p_2 and their counterparts in the other new-order instance lead to an unreorderable SC-cycle. To eliminate this unreorderable SC-cycle, ROCOCO must execute p_1 and p_2 as a single merged piece. In the start phase, the coordinator executes p_1 and p_2 using a three-phase OCC+2PC (execute-prepare-commit). If OCC+2PC aborts the coordinator retries until it succeeds. In the commit phase of OCC+2PC, each server will then add appropriate edges and vertexes to its *dep* graph, and reply with all undecided ancestor transactions in *dep*, as in the normal start phase of ROCOCO.

In our experience, simple workloads such as RUBiS[3] and Retwis[2] do not require merged pieces. TPC-C is much more complex. However, with the support of read-only transactions, there are no unreorderable SC-cycles in TPC-C and therefore no merged pieces.

5 Evaluation

Our evaluation explores two key questions:

1. How does the throughput and latency of ROCOCO compare to that of traditional approaches under varying levels of contention?
2. Can ROCOCO scale out with OLTP workloads?

This section will show that ROCOCO has higher throughput and lower latency than OCC and 2PL under all levels of contention and that as contention increases

ROCOCO's advantage increases. It will also show that ROCOCO scales near linearly in a complex workload, where contention rate grows as the system scales.

5.1 Implementation

We implemented a distributed in-memory key-value store with transactional support using ROCOCO. Our prototype contains over 20 000 lines of C++ code, of which 10 000 are for concurrency control. It uses a custom RPC library implemented by one of the authors for communication [4]. It adopts the simple threading model of H-Store [52] that uses a single worker thread on each server (core) to sequentially process the server's transaction pieces. The worker thread performs all blocking operations asynchronously. Currently, stored procedure—i.e., a piece of a transaction—is written as a C++ function that is loaded into the server binary at launch time.

2PL and OCC implementation. Our prototype also implements 2PL+2PC or OCC+2PC. Both protocols include an execute phase in which the coordinator instructs each involved server to execute a transaction piece and then a commit phase based on 2PC.

For 2PL, servers acquire locks during the execute phase. Subsequently, in the 2PC prepare phase, the coordinator instructs each involved server to durably log its buffered writes and lock acquisitions. In 2PC's commit phase, servers release locks and make writes visible. We use the wound-wait strategy [48], also used in Spanner[19], to avoid deadlocks.

For OCC, servers return the versions of data items read to the coordinator during the execute phase. In 2PC's prepare phase, each involved server acquires write locks, acquires read locks to validate the freshness of reads, and durably logs its writes and vote decisions. In 2PC's commit phase, servers release locks and make writes visible.

5.2 Experimental Setup

Unless otherwise mentioned, all experiments are conducted on the Kodiak testbed [1]. Each machine has a single-core 2.6GHz AMD Opteron 252 CPU with 8GB RAM and Gigabit Ethernet. Most experiments are bottlenecked on the server CPU. We have achieved much higher throughput when running on a local testbed with faster CPUs.

In all experiments, clients and servers run on different machines. Each client machine runs 1-30 single-threaded client processes while each server machine runs a single server process. Each data point in the graphs represents the median of at least five trials. Each trial is run for over 60s with the first and last quarter of each trial elided to avoid start up and cool down artifacts.

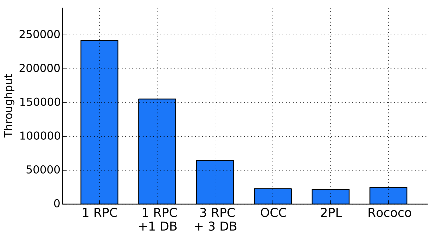


Figure 11: Throughput of baseline operations involving 3 servers. The transaction workload for OCC/2PL/ROCOCO has no contention.

Logging is turned off for all experiments because the Kodiak testbed does not include SSDs. We explore the overhead of logging to SSDs in our local testbed in Section 5.7. Logging always amplifies the throughput advantage of ROCOCO over 2PL and OCC. Logging sometimes increases the latency of ROCOCO over 2PL and OCC, but this is at most a few ms.

5.3 Micro-Benchmarks

To understand the base performance of our implementation, we ran a series of micro-benchmarks in a workload with *no contention*. The experiment uses three servers and its workload is a simple transaction that updates three counters, one on each server.

Figure 11 shows the throughput for a few baseline operations, from left to right, a null RPC to one of the servers (1 RPC), an RPC performing a database update at one of the servers (1 RPC+DB), three parallel RPCs each doing a database update at a different server (3 RPC+DB), and the simple transaction performing 3 database updates using OCC, 2PL, or ROCOCO.

Each server is able to handle ~75k null RPCs per second and is bottlenecked on CPU. The 1 RPC+DB throughput is slightly lower and is also bottlenecked on CPU, suggesting that the cost of a database access is relatively small compared to the cost of RPC. OCC and 2PL have similar throughput, roughly 1/3 of 3 RPC+DB, because they both require three rounds of RPCs to commit. ROCOCO requires two rounds of RPCs but incurs higher CPU cost to process dependency information, resulting in similar throughput to 2PL/OCC.

5.4 Scaled TPC-C Workload Overview

We evaluate ROCOCO's performance under contention using a scaled out version of the popular TPC-C [5] benchmark. This subsection explains how we scaled out TPC-C and how this differs from prior work.

Partition Strategy. Prior work partitioned the TPC-C database by warehouse [20, 54, 33, 55], with each

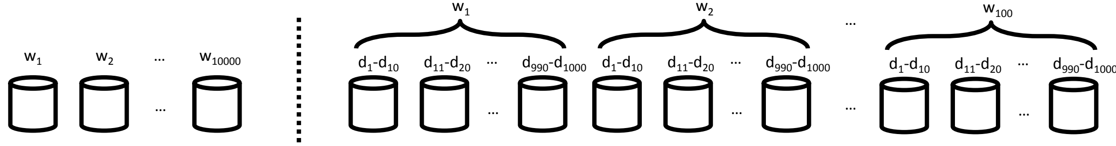


Figure 12: TPC-C sharding and scaling strategy. The left figure is the conventional strategy of scaling by increasing the number of warehouses. The right figure is our strategy of scaling inside a warehouse. w stands for warehouse, d stands for district.

server holding all data (including stock level and customer orders) related to a warehouse. This partition-by-warehouse strategy has two downsides. First, because only a single server handles each warehouse's data and requests, there is no performance scaling within a warehouse. This is acceptable in stock TPC-C that dictates a relatively low customer-to-warehouse ratio of 30K:1. However, in practice, a warehouse might need to handle a much larger population of users. For example, Amazon has >300 million customers served by ~100 warehouses. In these scenarios, the throughput of a single warehouse must scale beyond a single machine. Second, partition-by-warehouse does not stress the performance of distributed transactions, because only a minority (<15%) of all transactions involve more than one server.

To allow scaling within a warehouse, we partition the database by item or by district, of which there are many within a warehouse. Tables storing district related information are sharded according to `warehouse_id` and `district_id` (Figure 12). The stock level table is sharded by `warehouse_id` and `item_id`. We remove the `w_ytd` field that keeps track of the total value of purchases within a warehouse. To obtain the same information, we use a read-only transaction that reads the `d_ytd` value for each district and sums them up. This strategy avoids `w_ytd` from becoming a bottleneck for all new-orders within a warehouse. The original TPC-C benchmark uses a ratio of 30K:10:1 between customer, district and warehouse. We change it to 3M:1K:1 so that our ratio of customer-to-district remains the same as in the original TPC-C.

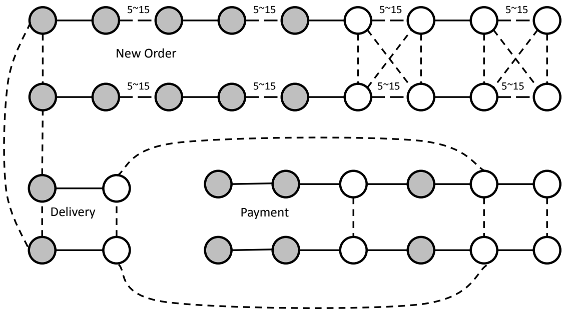


Figure 13: SC-graph for the TPC-C benchmark

Transaction pieces and the SC-graph. The TPC-C benchmark consists of five transactions: new_order, payment, delivery, order_status, and stock_level. Order_status, and stock_level are read-only transactions. Figure 13 shows the SC-graph for the remaining three transactions. The new order transaction contains five kinds of pieces, four of which occur 5 to 15 times, depending on how many items the transaction touches.

Table 1 shows the percentage of each transaction type in a random trial with ROCOCO, which matches the specified mix for TPC-C, and the average number of pieces included in each transaction.

	new-order	payment	order-status	delivery	stock-level
type	rw	rw	ro	rw	ro
ratio	44.97%	43.00%	4.03%	4.00%	4.00%
# pieces	40.97	4	3	40	210.93

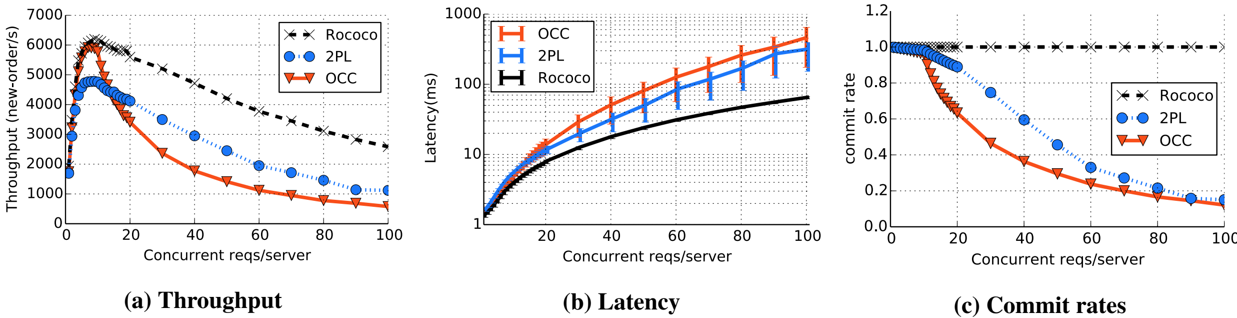
Table 1: TPC-C commit transaction mix ratio in a ROCOCO trial. rw stands for general read-write transactions and ro stands for read-only.

5.5 Contention

We ran the scaled TPC-C benchmark to explore how 2PL, OCC, and ROCOCO perform under varying levels of contention. Figure 14 shows the results of this experiment. Figure 14a shows the throughput; Figure 14b shows the median, 90th percentile, and 99th percentile latency; and Figure 14c shows the commit rate.

Experimental Parameters. We ran the contention experiment with 8 servers that each served 10 districts. All 80 districts belong to 1 warehouse. We vary the clients per server from 1 to 100 with each client issuing a mixture of TPC-C transactions according to the specification in a closed loop. When the number of clients is higher, there are more requests per server and thus higher contention.

The contention level is also affected by the number of districts per server. If a core serves too few districts—e.g., 1 district per server—OCC and 2PL are unable to saturate the server's CPU under low contention. This is because a core needs at least four clients to saturate its



(a) Throughput **(b) Latency** **(c) Commit rates**

Figure 14: New-order transaction characteristics in TPC-C mixed workload benchmark, with 8 servers. In the latency graph, the line shows the 90th percentile, and bars show the median and 99th percentile.

CPU, but four clients on a single district causes many conflicts. If a core serves too many districts—e.g., 100 districts per server—a large number of clients are needed to generate even moderate levels of contention. In order to explore varying levels of contention, we configure each core to serve 10 districts.

Minimal Contention. When the number of concurrent requests per server is fewer than 10, there is almost no contention in the system. OCC reaches its maximum throughput, 5916 new-orders/s, with ~7 clients per server (Figure 14a) when each server's CPU is saturated. 2PL performs similarly, but has a lower maximum throughput, 4781 new-orders/s, because of the overhead of maintaining the read/write lock queues. ROCOCO has the highest maximum throughput, 6197 new-orders/s, because of computational savings from its one fewer round of RPCs, which outweighs the computational cost of the graph computations it performs.

Low Contention. When the number of concurrent requests increases from 10 to 20 per server, the contention level increases from minimal to low. OCC is very sensitive to this increase in contention with a large performance drop to only half of its peak throughput. This drop in throughput comes from repeated aborts and retries in OCC as evidenced by the drop in its commit rate to ~60%. 2PL is less sensitive to the increase in contention because it always allows the oldest transaction to commit, which guarantees progress and limits the number of retries for a transaction. This is also observable from the commit rate, which drops by only ~10%. The median latency of 2PL and OCC both increase to about 20ms, due to the abort/retry. The performance of ROCOCO is relatively unaffected because it does not abort on read-write transactions. The median latency of ROCOCO increase to 10ms, due primarily to more transaction requests waiting in the message queue.

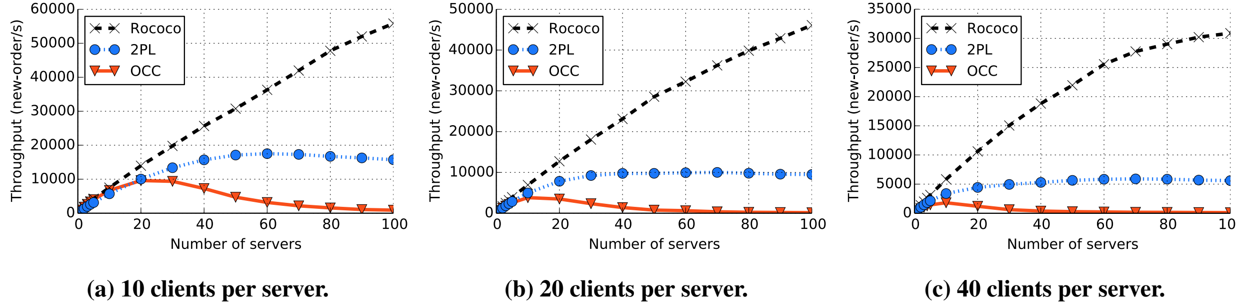
Moderate Contention. When the number of concurrent requests increases from 20 to 40 per server, the contention increases from low to moderate. OCC continues to be very sensitive to this continued increase in contention. With 40 concurrent requests per server, the throughput of OCC is 1774 new-orders/s, one third of its peak, and its 99th percentile latency is over 67ms. The performance of 2PL also starts to drop quickly under moderate contention. Its throughput drops to 2950 new-orders/s, half of its peak, and its 99th percentile latency increases to 38ms. ROCOCO is also affected by the increase to moderate contention, though it is less sensitive than OCC and 2PL because it avoids aborting and retrying transactions. Its throughput drops by 24%, and its 99th percentile latency increases to 12ms.

High Contention. When the number of concurrent requests increases to over 40 per server, the benchmark reflects a high-level of contention. In the worst case, the throughput of OCC drops to a few hundred, due to large amounts of aborts and retries, with its commit rate dropping to 16%. 2PL has better performance than OCC, especially as measured by latency, because its wound-wait strategy ensures progress. But, 2PL's throughput and commit rate decrease significantly because of the large number of aborts. ROCOCO demonstrates the best performance with high contention. Its throughput drops to only 2584 new-orders/s, which is 130% higher than 2PL and 347% higher than OCC. More importantly, because ROCOCO avoids aborting and retrying, its latency is only 10%-40% of that of OCC and 2PL.

5.6 Scalability

We evaluate the scalability of ROCOCO in two different ways. The first is conventional TPC-C scaling by increasing the number of warehouses with a fixed number of districts per warehouse. In this case all protocols scale linearly (not shown) because each added warehouse is almost entirely independent of the existing warehouses and the contention rate—i.e., how frequently different transactions interact—remains constant.

The second, and more representative, experiment is scaling inside a warehouse from 10 districts on 1 server



(a) 10 clients per server. (b) 20 clients per server. (c) 40 clients per server.

Figure 15: Scaling across districts

to 1000 districts on 100 servers. Scaling the number of districts increases the contention rate, which we believe is meaningful because as a system scale it is more likely that transactions will interact. For instance, in an e-commerce site, as the site becomes more popular it is likely to gain many more new customers than new items. In addition, a small set of items tend to be very popular and becomes more and more likely than have different customer try to concurrently purchase them, which is a source of contention in TPC-C new order transactions.

Our experiments use three levels of contention; low contention with 10 clients per server, shown in Figure 15a; moderate contention with 20 clients per server, shown in Figure 15b; and high contention with 40 clients per server, shown in Figure 15c.

ROCOCO scales near linearly when contention is low, with its throughput increasing from 7519 new-orders/s with 10 servers, to 13971 with 20 server, 25671 with 40 servers, and 47787 with 80 servers. The throughput of OCC and 2PL are far lower. OCC peaks at 9611 new-orders/s with 20 servers and 2PL peaks at 17521 with 60 servers. OCC and 2PL do not scale well because the increasing contention rate leads to more aborts.

ROCOCO also scales near linearly when contention is moderate, with its throughput increasing from 6921 new-orders/s with 10 servers, to 12736 with 20 server, 23117 with 40 servers, and 39853 with 80 servers. The higher levels of contention quickly lead to high abort rate for OCC and 2PL, which peak at 3816 and 10005 new-orders/s respectively.

When contention is high with 40 clients per server, ROCOCO still scales well, though this scaling is no longer near linear. The scaling is not linear because at this high level of contention ROCOCO propagates and processes much larger dependency graphs.

5.7 Logging to SSDs

This subsection explores the effect of synchronous logging in 2PL, OCC, and ROCOCO. This experiment is conducted in our local cluster that is equipped with SSDs, all other experiments were performed on the Kodiak cluster. To ensure that the log is safely persisted we turned off caching in the operating system and disks. We call `fsync` and wait for its return before we consider the log to be successfully written. We use a batch time of ~1ms before each `fsync`, which increases throughput significantly at the cost of slightly higher latency.

Table 2 shows the performance with 8 servers and 20 concurrent clients per server. 2PL and ROCOCO both have about 20% throughput drop and a latency increase of 2-3ms. The performance of OCC is more severely impacted as the batched logging resulting in requests holding their locks for longer in the prepare phase, which increases the likelihood of aborts. This effect is evident in the decreased commit rate for OCC.

	Throughput (new-orders/s)	Commit Rate (%)	Latency(ms) 50%	90%	99%
OCC					
no log	4109	63.82	8.49	11.35	13.60
logging	2748	54.28	12.17	18.35	22.79
2PL					
no log	4944	88.52	8.63	10.20	11.29
logging	4038	88.76	10.89	13.01	14.48
ROCOCO					
no log	6464	100	6.52	7.12	7.33
logging	5382	100	8.78	9.62	9.94

Table 2: Effect of logging in our local cluster

6 Related Work

General transactions with 2PL and OCC. Many seminal distributed databases such as Gamma [22], Bubba [16], and R* [45] use forms of 2PL. Spanner [19] is Google's linearizable global-scale database that uses 2PL for read-write transactions and a separate timestamp based protocol from read-only transactions. Replicated Commit optimizes the across site latency in Spanner's commit protocol [44].

OCC is also used in several recent systems, such as H-Store [33] and VoltDB [6]. MDCC [35] uses OCC for geo-replicated storage. Percolator uses OCC to pro-

vide snapshot isolation [14]. Adya et al. [7] use loosely synchronized clocks and timestamps in the validation of OCC.

Observations have been made that OCC and 2PL behave well with no contention, but the performance will drop quickly as contention increases [8, 30]. This is also what we have observed in our evaluation.

Concurrency control with limited transactions. A recent trend is to improve performance by supporting limited types of distributed transactions. For example, MegaStore [10] only provides serializable transactions within a data partition. Other systems, such as Granola [20], Calvin [54] and Sinfonia [9] propose concurrency control protocols for transactions with known read-/write keys.

Sinfonia's protocol is based on OCC and 2PC. Granola achieves a deterministic serial order of conflicting transactions by exchanging timestamp between servers while Calvin achieves this by using a separate sequencing layer that assigns all transactions to a deterministic locking order to ensure isolation at each participating server. None of these systems supports key-dependent transactions: the read/write sets must be known apriori. In contrast, ROCOCO does support such transactions with immediate pieces.

Dependency and interference. Our work is motivated by recent efforts on consensus protocols such as Generalized-Paxos [38] and EPaxos [46], which uses dependency to reorder interfering commands in state machine replication (SMR). Paxos addresses consistent data replication and is used as a black-box module to provide replication in databases. However, consensus protocols bear some resemblance to distributed transaction protocols because reaching consensus is similar to committing a write-only transactions between several replicas of the same item [29].

COPS/Eiger [42, 43] track dependency between operations to provide causal+ consistency in geo-replicated key-value stores. Dependencies are also used to provide read-only/write-only transaction support. Warp [24] is a transaction layer on top of HyperDex [23] and its protocol also performs dependency tracking.

A major difference between ROCOCO and the above dependency-tracking systems is that ROCOCO can avoid aborts for transactions that require intermediate results between pieces. ROCOCO pushes this boundary using offline checking to eliminate possible unreorderable interleavings and by tracking finer grained dependencies to break dependency cycles in a serializable way.

Transaction decomposition and offline checking. The database community has explored various aspects of decomposing a transaction into smaller pieces for improved performance. [27, 11, 17, 25] Shasha et al. [50, 49] propose the theory of transaction chopping which uses SC-cycles to analyze possible conflicts that may lead to non-serializable execution. Lynx [57] uses transaction chopping and chain execution to achieve serializability and low latency simultaneously in a geo-distributed system. It uses commutative operations and origin ordering to ensure SC-cycles in web applications are safe. Compared to Lynx, ROCOCO distinguishes reorderable SC-cycles from unreorderable ones, executes pieces in parallel, and supports the strict form of serializability.

Geo-distributed systems with weaker semantics. Geo-distributed systems face a tradeoff between strong semantics and low latency. Systems such as Dynamo [21] and Cassandra [37] embrace latency and offer eventual consistency without transactional support. PNUTS [18] offers per-record timeline consistency. Walter provides parallel snapshot isolation [51] and Gemini provides Red/Blue consistency [40]. ROCOCO supports transactions with the strongest semantics (i.e. strict serializability) and thus will incur cross-datacenter latency when running in a geo-distributed setting.

7 Conclusion

This paper presented ROCOCO, a novel concurrency control protocol for distributed transactions. With the help of offline checking, ROCOCO reorders pieces of interfering transactions into a strict-serializable order and avoids aborts. In a scaled TPC-C benchmark ROCOCO outperformed conventional protocols and showed stable performance with increasing contention.

Acknowledgement

This work is supported in part by the National Science Foundation under award CNS-1218117. We also thank Garth Gibson and the PRObE team for the testbed (NSF awards CNS-1042537 and CNS-1042543).

Shuai Mu's work is also supported by the China Scholarship Council.

References

[1] Kodiak testbeds. http://portal.nmc-probe.org/.

[2] Retwis. http://retwis.antirez.com/.

[3] RUBiS. http://rubis.ow2.org/.

[4] Simple RPC in C++. https://github.com/santazhang/simple-rpc.

[5] TPC-C Benchmark. http://www.tpc.org/tpcc/.

[6] VoltDB. http://www.voltdb.com/.

[7] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 23–34, New York, NY, USA, 1995. ACM.

[8] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems (TODS)*, 12(4):609–654, 1987.

[9] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[10] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, volume 11, pages 223–234, 2011.

[11] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis. Concurrency control for step-decomposed transactions. *Information Systems*, 24(8):673–698, 1999.

[12] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[14] P. Bhatotia, A. Wieder, İ. E. Akkuş, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, pages 18–18. USENIX Association, 2011.

[15] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 11–11. USENIX Association, 2011.

[16] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):4–24, 1990.

[17] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–239, 1992.

[18] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[20] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 21–21. USENIX Association, 2012.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[22] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):44–62, 1990.

[23] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[24] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight Multi-Key Transactions for Key-Value Stores. Technical report, 2014.

[25] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems (TODS)*, 8(2):186–213, 1983.

[26] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. *Computers, IEEE Transactions on*, 100(5):391–399, 1984.

[27] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 249–259, New York, NY, USA, 1987. ACM.

[28] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509–516, 1992.

[29] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[30] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992. ACM, 2008.

[31] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[33] E. P. Jones, D. J. Abadi, and S. Madden. Low over-

head concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 603–614. ACM, 2010.

[34] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[35] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[36] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[37] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[38] L. Lamport. Generalized consensus and paxos.

[39] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[40] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2012.

[41] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of the first international symposium on Databases in parallel and distributed systems*, pages 177–187. IEEE Computer Society Press, 2000.

[42] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Symposium on Networked Systems Design and Implementation*, 2013.

[44] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672, 2013.

[45] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[46] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in Egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.

[47] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting More Concurrency from Distribted Transactions. Technical Report TR2014-970, New York University, Courant Institute of Mathematical Sciences, 2014.

[48] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)*, 3(2):178–198, 1978.

[49] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[50] D. Shasha, E. Simon, and P. Valduriez. Simple rational guidance for chopping up transactions. In *ACM SIGMOD Record*, volume 21, pages 298–307. ACM, 1992.

[51] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[52] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[53] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[54] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[55] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[56] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, sql or C++. In *Seventh International Workshop on High Performance Transaction Systems, Asilomar*, 1997.

[57] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.

Salt: Combining ACID and BASE in a Distributed Database

Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang,
Navid Yaghmazadeh, Lorenzo Alvisi, Prince Mahajan

The University of Texas at Austin

Abstract: This paper presents Salt, a distributed database that allows developers to improve the performance and scalability of their ACID applications through the incremental adoption of the BASE approach. Salt's motivation is rooted in the Pareto principle: for many applications, the transactions that actually test the performance limits of ACID are few. To leverage this insight, Salt introduces *BASE transactions*, a new abstraction that encapsulates the workflow of performance-critical transactions. BASE transactions retain desirable properties like atomicity and durability, but, through the new mechanism of *Salt Isolation*, control which granularity of isolation they offer to other transactions, depending on whether they are BASE or ACID. This flexibility allows BASE transactions to reap the performance benefits of the BASE paradigm without compromising the guarantees enjoyed by the remaining ACID transactions. For example, in our MySQL Cluster-based implementation of Salt, BASE-ifying just one out of 11 transactions in the open source ticketing application Fusion Ticket yields a 6.5x increase over the throughput obtained with an ACID implementation.

1 Introduction

This paper presents *Salt*, a distributed database that, for the first time, allows developers to reap the complementary benefits of both the ACID and BASE paradigms within a single application. In particular, Salt aims to dispel the false dichotomy between performance and ease of programming that fuels the ACID vs. BASE argument.

The terms of this debate are well known [28, 30, 37]. In one corner are ACID transactions [7–9, 12–14, 36]: through their guarantees of Atomicity, Consistency, Isolation, and Durability, they offer an elegant and powerful abstraction for structuring applications and reasoning about concurrency, while ensuring the consistency of the database despite failures. Such ease of programming, however, comes at a significant cost to performance and availability. For example, if the database is distributed, enforcing the ACID guarantees typically requires running a distributed commit protocol [31] for each transaction while holding an exclusive lock on *all* the records modified during the transaction's entire execution.

In the other corner is the BASE approach (Basically-Available, Soft state, Eventually consistent) [28, 32, 37], recently popularized by several NoSQL systems [1, 15, 20, 21, 27, 34]. Unlike ACID, BASE offers more of a set of programming guidelines (such as the use of *partition local transactions* [32, 37]) than a set of rigorously specified properties and its instantiations take a variety of application-specific forms. Common among them, however, is a programming style that avoids distributed transactions to eliminate the performance and availability costs of the associated distributed commit protocol. Embracing the BASE paradigm, however, exacts its own heavy price: once one renounces ACID guarantees, it is up to developers to explictly code in their applications the logic necessary to ensure consistency in the presence of concurrency and faults. The complexity of this task has sparked a recent backlash against the early enthusiasm for BASE [22, 38]—as Shute et al. put it "Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains" [38].

Salt aims to reclaim most of those performance gains while keeping complexity in check. The approach that we propose to resolve this tension is rooted in the Pareto principle [35]. When an application outgrows the performance of an ACID implementation, it is often because of the needs of only a handful of transactions: most transactions never test the limits of what ACID can offer. Numerous applications [2, 4, 5, 10, 11] demonstrate this familiar lopsided pattern: few transactions are performance-critical, while many others are either lightweight or infrequent; e.g. administrative transactions. Our experience confirms this pattern. For example, running the TPC-C benchmark [23] on a MySQL cluster, we found that, as the load increases, only two transactions take much longer to complete—a symptom of high contention; other transactions are unaffected. Similarly, we found that the ACID throughput of Fusion Ticket [6], a popular open source online ticketing application that uses MySQL as its backend database, is limited by the performance of just *one* transaction out of 11. It is tempting to increase the concurrency of those transactions by splitting them into smaller ones. Doing so, however, exposes fundamental limitations of the ACID paradigm.

One of the main attractions of the ACID paradigm is to pack in a single abstraction (the ACID transaction) the four properties that give ACID its name. This tight cou-

pling of all four properties, however, comes at the cost of little flexibility. In particular, offering atomicity and isolation at the same granularity is the very reason why ACID transactions are ill-equipped to manage effectively the tradeoff between performance and ease of programming. First, splitting an ACID transaction into several smaller transactions to increase concurrency would of course result in the loss of the all-or-nothing atomicity guarantees of the original transaction. But even more disturbing would be the resulting loss in isolation, not only for the split transaction, but for *all* transactions in the system: any transaction would be able to indiscriminately access what used to be intermediate database states protected by the guarantees of isolation, making it much harder to reason about correctness. Nonetheless, this is, in essence, the strategy adopted by the BASE approach, which for good measure also gives up consistency and durability in the bargain.

Motivated by these insights, our vision for Salt is simple: to create a database where the ACID and BASE paradigms can safely coexist within the same application. In particular, Salt enables ACID applications that struggle to meet their growing performance demands to improve their availability and scalability by incrementally "BASE-ifying" only the few ACID transactions that are performance-critical, without compromising the ACID guarantees enjoyed by the remaining transactions.

Of course, naively BASE-ifying selected ACID transactions may void their atomicity guarantees, compromise isolation by exposing intermediate database states that were previously unobservable, and violate the consistency invariants expected by the transactions that have not been BASE-ified. To enable mutually beneficial coexistence between the ACID and BASE paradigms, Salt introduces a new abstraction: *BASE transactions*.

BASE transactions loosen the tight coupling between atomicity and isolation enforced by the ACID paradigm to offer a unique combination of features: the performance and availability benefits of BASE-style partition-local transactions together with the ability to express and enforce atomicity at the granularity called for by the application semantics.

Key to this unprecedented combination is *Salt Isolation*, a new isolation property that regulates the interactions between ACID and BASE transactions. For performance, Salt Isolation allows concurrently executing BASE transactions to observe, at well-defined spots, one another's internal states, but, for correctness, it completely prevents ACID transactions from doing the same.

We have built a Salt prototype by modifying an ACID system, the MySQL Cluster distributed database [9], to support BASE transactions and Salt Isolation. Our initial evaluation confirms that BASE transactions and Salt Isolation together allow Salt to break new ground in balancing performance and ease of programming.

In summary, we make the following contributions:

- We introduce a new abstraction, *BASE transactions*, that loosens the tight coupling between atomicity and isolation to reap the performance of BASE-style applications, while at the same time limiting the complexity typically associated with the BASE paradigm.
- We present a novel isolation property, *Salt Isolation*, that controls how ACID and BASE transactions interact. Salt Isolation allows BASE transactions to achieve high concurrency by observing each other's internal states, without affecting the isolation guarantees of ACID transactions.
- We present an evaluation of the Salt prototype, that supports the view that combining the ACID and BASE paradigms can yield high performance with modest programming effort. For example, our experiments show that, by BASE-ifying just one out of 11 transactions in the open source ticketing application Fusion Ticket, Salt's performance is *6.5x* higher than that of an ACID implementation.

The rest of the paper proceeds as follows. Section 2 sheds more light on the ACID vs. BASE debate, and the unfortunate tradeoff it imposes on developers, while Section 3 proposes a new alternative, Salt, that sidesteps this tradeoff. Section 4 introduces the notion of BASE transactions and Section 5 presents the novel notion of Salt Isolation, which allows ACID and BASE transactions to safely coexist within the same application. Section 6 discusses the implementation of our Salt prototype, Section 7 shows an example of programming in Salt, and Section 8 presents the results of our experimental evaluation. Section 9 discusses related work and Section 10 concludes the paper.

2 A stark choice

The evolution of many successful database applications follows a common narrative. In the beginning, they typically rely on an ACID implementation to dramatically simplify and shorten code development and substantially improve the application's robustness. All is well, until success-disaster strikes: the application becomes wildly popular. As the performance of their original ACID implementation increasingly proves inadequate, developers are faced with a Butch Cassidy moment [33]: holding their current ground is untenable, but jumping off the cliff to the only alternative—a complete redesign of their application following the BASE programming paradigm—is profoundly troubling. Performance may soar, but so will complexity, as all the subtle issues that ACID handled automatically, including error handling, concurrency control, and the logic needed for consistency enforcement, now need to be coded explicitly.

```
1   // ACID transfer transaction
2   begin transaction
3     Select bal into @bal from accnts where id = sndr
4     if (@bal >= amt)
5       Update accnts set bal −= amt where id = sndr
6       Update accnts set bal += amt where id = rcvr
7   commit

9   // ACID total−balance transaction
10  begin transaction
11    Select sum(bal) from accnts
12  commit
```

(a) The ACID approach.

```
1   // transfer using the BASE approach
2   begin local−transaction
3     Select bal into @bal from accnts where id = sndr
4     if (@bal >= amt)
5       Update accnts set bal −= amt where id = sndr
6       // To enforce atomicity, we use queues to communicate
7       // between partitions
8       Queue message(sndr, rcvr, amt) for partition(accnts, rcvr)
9   end local−transaction

11  // Background thread to transfer messages to other partitions
12  begin transaction // distributed transaction to transfer queued msgs
13    <transfer messages to rcvr>
14  end transaction

16  // A background thread at each partition processes
17  // the received messages
18  begin local−transaction
19    Dequeue message(sndr, rcvr, amt)
20    Select id into @id from accnts where id = rcvr
21    if (@id ≠ ∅) // if rcvr's account exists in database
22      Update accnts set bal += amt where id = rcvr
23    else // rollback by sending the amt back to the original sender
24      Queue message(rcvr, sndr, amt) for partition(accnts, sndr)
25  end local−transaction

27  // total−balance using the BASE approach
28  // The following two lines are needed to ensure correctness of
29  // the total−balance ACID transaction
30  <notify all partitions to stop accepting new transfers>
31  <wait for existing transfers to complete>
32  begin transaction
33    Select sum(bal) from accnts
34  end transaction
35  <notify all partitions to resume accepting new transfers>
```

(b) The BASE approach.

Fig. 1: A simple banking application with two implementations: (a) ACID and (b) BASE

Figure 1 illustrates the complexity involved in transitioning a simple application from ACID to BASE. The application consists of only two transactions, `transfer` and `total-balance`, accessing the *accnts* relation. In the original ACID implementation, the transfer either commits or is rolled-back automatically despite failures or invalid inputs (such as an invalid *rcvr* id), and it is easy to add constraints (such as $bal \geq amt$) to ensure consistency invariants. In the BASE implementation, it is instead up to the application to ensure consistency and atomicity despite failures that occur between the first and second transaction. And while the level of isolation (the property that specifies how and when changes to the database performed by one transaction become visible to transactions that are executing concurrently) offered by ACID transactions ensures that `total-balance` will compute accurately the sum of balances in *accnts*, in BASE the code needs to prevent explicitly (lines 30 and 31 of Figure 1(b)) `total-balance` from observing the intermediate state after the *sndr* account has been charged but before the *rcvr*'s has been credited.

It speaks to the severity of the performance limitations of the ACID approach that application developers are willing to take on such complexity.

The ACID/BASE dichotomy may appear as yet another illustration of the "no free lunch" adage: if you want performance, you must give something up. Indeed—but BASE gives virtually *everything* up: the entire application needs to be rewritten, with no automatic support for either atomicity, consistency, or durability, and with isolation limited only to partition-local transactions. Can't we aim for a more reasonable bill?

3 A grain of Salt

One ray of hope comes, as we noted in the Introduction, from the familiar Pareto principle: even in applications that outgrow the performance achievable with ACID solutions, not all transactions are equally demanding. While a few transactions require high performance, many others never test the limits of what ACID can offer. This raises an intriguing possibility: could one identify those few performance-critical transactions (either at application-design time or through profiling, if an ACID implementation of the application already exists) and somehow only need to go through the effort of BASE-ifying *those* transactions in order to get most of the performance benefits that come from adopting the BASE paradigm?

Realizing this vision is not straightforward. For example, BASE-ifying only the `transfer` transaction in the simple banking application of Figure 1 would allow `total-balance` to observe a state in which *sndr* has been charged but *rcvr*'s has not yet been credited, causing it to compute incorrectly the bank's holdings. The central issue is that BASE-ifying transactions, even if only a few, can make suddenly accessible to all transactions what previously were invisible intermediate database states. Protecting developers from having to worry about such intermediate states despite failures and concurrency, however, is at the core of the ease of programming offered by the transactional programming paradigm. Indeed, quite naturally, isolation (which regulates which states can be accessed when transactions execute concurrently) and atomicity (which frees from worrying about intermediate states during failures) are typically offered at the same granularity—that of the ACID transaction.

We submit that while this tight coupling of atomicity and isolation makes ACID transactions both powerful and attractively easy to program with, it also limits their ability to continue to deliver ease of programming when

performance demands increase. For example, splitting an ACID transaction into smaller transactions can improve performance, but at the cost of shrinking the original transaction's guarantees in terms of both atomicity and isolation: the all-or-nothing guarantee of the original transaction is unenforceable on the set of smaller transactions, and what were previously intermediate states can suddenly be accessed indiscriminately by all other transactions, making it much harder to reason about the correctness of one's application.

The approach that we propose to move beyond today's stark choices is based on two propositions: first, that the coupling between atomicity and isolation should be loosened, so that providing isolation at a fine granularity does not necessarily result in shattering atomicity; and second, that the choice between either enduring poor performance or allowing indiscriminate access to intermediate states by all transactions is a false one: instead, complexity can be tamed by giving developers control over who is allowed to access these intermediate states, and when.

To enact these propositions, the Salt distributed database introduces a new abstraction: *BASE transactions*. The design of BASE transactions borrows from nested transactions [41], an abstraction originally introduced to offer, for long-running transactions, atomicity at a finer granularity than isolation. In particular, while most nested transaction implementations define isolation at the granularity of the parent ACID transaction,[1] they tune the mechanism for enforcing atomicity so that errors that occur within a nested subtransaction do not require undoing the entire parent transaction, but only the affected subtransaction.

Our purpose in introducing BASE transactions is similar in spirit to that of traditional nested transactions: both abstractions aim at gently loosening the coupling between atomicity and isolation. The issue that BASE transactions address, however, is the flip side of the one tackled by nested transactions: this time, the challenge is to provide *isolation* at a finer granularity, without either drastically escalating the complexity of reasoning about the application, or shattering atomicity.

4 BASE transactions

Syntactically, a BASE transaction is delimited by the familiar `begin BASE transaction` and `end BASE transaction` statements. Inside, a BASE transaction contains a sequence of *alkaline* subtransactions—nested transactions that owe their name to the novel way in which they straddle the ACID/BASE divide.

When it comes to the granularity of atomicity, as we will see in more detail below, a BASE transaction provides the same flexibility of a traditional nested transaction: it can limit the effects of a failure within a single alkaline subtransaction, while at the same time it can ensure that the set of actions performed by all the alkaline subtransactions it includes is executed atomically. Where a BASE transaction fundamentally differs from a traditional nested transaction is in offering *Salt Isolation*, a new isolation property that, by supporting multiple granularities of isolation, makes it possible to control which internal states of a BASE transaction are externally accessible, and by whom. Despite this unprecedented flexibility, Salt guarantees that, when BASE and ACID transactions execute concurrently, ACID transactions retain, with respect to all other transactions (whether BASE, alkaline, or ACID), the same isolation guarantees they used to enjoy in a purely ACID environment. The topic of how Salt isolation supports ACID transactions across all levels of isolation defined in the ANSI/ISO SQL standard is actually interesting enough that we will devote the entire next section to it. To prevent generality from obfuscating intuition, however, the discussion in the rest of this section assumes ACID transactions that provide the popular *read-committed* isolation level.

Independent of the isolation provided by ACID transactions, a BASE transaction's basic unit of isolation are the alkaline subtransactions it contains. Alkaline subtransactions retain the properties of ACID transactions: in particular, when it comes to isolation, no transaction (whether ACID, BASE or alkaline) can observe intermediate states produced by an uncommitted alkaline subtransaction. When it comes to observing the state produced by a *committed* alkaline subtransaction, however, the guarantees differ depending on the potential observer.

- The committed state of an alkaline subtransaction is observable by other BASE or alkaline subtransactions. By leveraging this finer granularity of isolation, BASE transactions can achieve levels of performance and availability that elude ACID transactions. At the same time, because alkaline subtransactions are isolated from each other, this design limits the new interleavings that programmers need to worry about when reasoning about the correctness of their programs: the only internal states of BASE transactions that become observable are those at the boundaries between its nested alkaline subtransactions.
- The committed state of an alkaline subtransaction is *not* observable by other ACID transactions until the parent BASE transaction commits. The internal state of a BASE transaction is then completely opaque to ACID transactions: to them, a BASE transaction looks

[1] *Nested top-level transactions* are a type of nested transactions that instead commit or abort independently of their parent transaction. They are seldom used, however, precisely because they violate the isolation of the parent transaction, making it hard to reason about consistency invariants.

```
1  // BASE transaction: transfer
2  begin BASE transaction
3    try
4      begin alkaline−subtransaction
5        Select bal into @bal from accnts where id = sndr
6        if (@bal >= amt)
7          Update accnts set bal −= amt where id = sndr
8      end alkaline−subtransaction
9    catch (Exception e) return // do nothing
10   if (@bal < amt) return // constraint violation
11   try
12     begin alkaline−subtransaction
13       Update accnts set bal += amt where id = rcvr
14     end alkaline−subtransaction
15   catch (Exception e) //rollback if rcvr not found or timeout occurs
16     begin alkaline−subtransaction
17       Update accnts set bal += amt where id = sndr
18     end alkaline−subtransaction
19 end BASE transaction

21 // ACID transaction: total−balance (unmodified)
22 begin transaction
23   Select sum(bal) from accnts
24 commit
```

Fig. 2: A Salt implementation of the simple banking application

just like an ordinary ACID transaction, leaving their correctness unaffected.

To maximize performance, we expect that alkaline subtransactions will typically be partition-local transactions, but application developers are free, if necessary to enforce critical consistency conditions, to create alkaline subtransactions that touch multiple partitions and require a distributed commit.

Figure 2 shows how the simple banking application of Figure 1 might look when programmed in Salt. The first thing to note is what has *not* changed from the simple ACID implementation of Figure 1(a): Salt does not require any modification to the ACID `total-balance` transaction; only the performance-critical `transfer` operation is expressed as a new BASE transaction. While the complexity reduction may appear small in this simple example, our current experience with more realistic applications (such as Fusion Ticket, discussed in Section 8) suggests that Salt can achieve significant performance gains while leaving untouched most ACID transactions. Figure 2 also shows another feature of alkaline subtransactions: each is associated with an exception, which is caught by an application-specific handler in case an error is detected. As we will discuss in more detail shortly, Salt leverages the exceptions associated with alkaline subtransactions to guarantee the atomicity of the BASE transactions that enclose them.

There are two important events in the life of a BASE transaction: *accept* and *commit*. In the spirit of the BASE paradigm, BASE transactions, as in Lynx [44], are accepted as soon as their first alkaline subtransaction commits. The atomicity property of BASE transactions ensures that, once accepted, a BASE transaction will eventually commit, i.e., all of its operations will have successfully executed (or bypassed because of some exception) and their results will be persistently recorded.

To clarify further our vision for the abstraction that BASE transactions provide, it helps to compare their guarantees with those provided by ACID transactions

Atomicity Just like ACID transactions, BASE transactions guarantee that *either all the operations they contain will occur, or none will*. In particular, atomicity guarantees that all accepted BASE transactions will eventually commit. Unlike ACID transactions, BASE transactions can be aborted only if they encounter an error (such as a constraint violation or a node crash) *before* the transaction is accepted. Errors that occur after the transaction has been accepted do not trigger an automatic rollback: instead, they are handled using exceptions. The details of our Salt's implementation of atomicity are discussed in Section 6.

Consistency Chasing higher performance by splitting ACID transactions can increase exponentially the number of interleavings that must be considered when trying to enforce integrity constraints. Salt drastically reduces this complexity in two ways. First, Salt does not require all ACID transactions to be dismembered: non-performance-critical ACID transactions can be left unchanged. Second, Salt does not allow ACID transactions to observe states inside BASE transactions, cutting down significantly the number of possible interleavings.

Isolation Here, BASE and ACID transactions depart, as BASE transactions provide the novel *Salt Isolation* property, which we discuss in full detail in the next section. Appealingly, Salt Isolation on the one hand allows BASE transactions to respect the isolation property offered by the ACID transactions they may execute concurrently with, while on the other yields the opportunity for significant performance improvements. In particular, under Salt Isolation a BASE transaction BT appears to an ACID transaction just like another ACID transaction, but other BASE transactions can observe the internal states that exist at the boundaries between adjacent alkaline subtransactions in BT.

Durability BASE transactions provide the same durability property of ACID transactions and of many existing NoSQL systems: *Accepted BASE transactions are guaranteed to be durable*. Hence, developers need not worry about losing the state of accepted BASE transactions.

5 Salt isolation

Intuitively, our goal for Salt isolation is to allow BASE transactions to achieve high degrees of concurrency, while ensuring that ACID transactions enjoy well-defined isolation guarantees. Before taking on this challenge in earnest, however, we had to take two important preliminary steps.

The first, and the easiest, was to pick the concurrency control mechanism on which to implement Salt

Isolation level	$\mathcal{L}$	$\mathcal{S}$
read-uncommitted	W	W
read-committed	W	R,W
repeatable-read	R,W	R,W
serializable	R,RR,W	R,RR,W

Table 1: Conflicting type sets $\mathcal{L}$ and $\mathcal{S}$ for each of the four ANSI isolation levels. R = Read, RR = Range Read, W = Write.

	ACID-R	ACID-W	alka-R	alka-W	saline-R	saline-W
ACID-R	✓	✗	✓	✗	✓	✗
ACID-W	✗	✗	✗	✗	✗	✗
alka-R	✓	✗	✓	✗	✓	✓
alka-W	✗	✗	✗	✗	✓	✓
saline-R	✓	✗	✓	✓	✓	✓
saline-W	✗	✗	✓	✓	✓	✓

Table 2: Conflict table for ACID, alkaline, and saline locks.

isolation. Our current design focuses on lock-based implementations rather than, say, optimistic concurrency control, because locks are typically used in applications that experience high contention and can therefore more readily benefit from Salt; also, for simplicity, we do not currently support multiversion concurrency control and hence snapshot isolation. However, there is nothing about Salt isolation that fundamentally prevents us from applying it to other mechanisms beyond locks.

The second step proved much harder. We had to crisply characterize what are exactly the isolation guarantees that we want our ACID transactions to provide. This may seem straightforward, given that the ANSI/ISO SQL standard already defines the relevant four isolation levels for lock-based concurrency: read-uncommitted, read-committed, repeatable read, and serializable. Each level offers stronger isolation than the previous one, preventing an increasingly larger prefix of the following sequence of undesirable phenomena: dirty write, dirty read, non-repeatable read, and phantom [18].

Where the challenge lies, however, is in preventing this diversity from forcing us to define four distinct notions of Salt isolation, one for each of the four ACID isolation levels. Ideally, we would like to arrive at a single, concise characterization of isolation in ACID systems that somehow captures all four levels, which we can then use to specify the guarantees of Salt isolation.

The key observation that ultimately allowed us to do so is that all four isolation levels can be reduced to a simple requirement: if two *operations* in different transactions[2] conflict, then the temporal dependency that exists between the earlier and the later of these operations must extend to the entire *transaction* to which the earlier operation belongs. Formally:

Isolation. *Let Q be the set of operation types $\{$read, range-read, write$\}$ and let $\mathcal{L}$ and $\mathcal{S}$ be subsets of Q. Further, let o_1 in txn_1 and o_2 in txn_2, be two operations, respectively of type $T_1 \in \mathcal{L}$ and $T_2 \in \mathcal{S}$, that access the same object in a conflicting (i.e. non read-read) manner. If o_1 completes before o_2 starts, then txn_1 must decide before o_2 starts.*

With this single and concise formulation, each of the ACID isolation levels can be expressed by simply instantiating appropriately $\mathcal{L}$ and $\mathcal{S}$. For example, $\mathcal{L} = \{write\}$ and $\mathcal{S} = \{read, write\}$ yields read-committed isolation. Table 1 shows the conflicting sets of operation types for all four ANSI isolation levels. For a given $\mathcal{L}$ and $\mathcal{S}$, we will henceforth say that two transactions are *isolated* from each other when Isolation holds between them.

Having expressed the isolation guarantees of ACID transactions, we are ready to tackle the core technical challenge ahead of us: defining an isolation property for BASE transactions that allows them to harmoniously coexist with ACID transactions. At the outset, their mutual affinity may appear dubious: to deliver higher performance, BASE transactions need to expose intermediate uncommitted states to other transactions, potentially harming Isolation. Indeed, the key to Salt isolation lies in controlling which, among BASE, ACID, and alkaline subtransactions, should be exposed to what.

5.1 The many grains of Salt isolation

Our formulation of Salt isolation leverages the conciseness of the Isolation property to express its guarantees in a way that applies to all four levels of ACID isolation.

Salt Isolation. *The Isolation property holds as long as (a) at least one of txn_1 and txn_2 is an ACID transaction or (b) both txn_1 and txn_2 are alkaline subtransactions.*

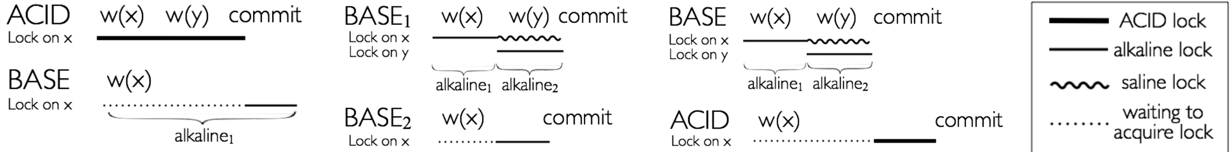
Informally, Salt isolation enforces the following constraint gradation:

- ACID transactions are isolated from all other transactions.
- Alkaline subtransactions are isolated from other ACID and alkaline subtransactions.
- BASE transactions expose their intermediate states (i.e. states produced at the boundaries of their alkaline subtransactions) to every other BASE transaction.

Hence, despite its succinctness, Salt isolation must handle quite a diverse set of requirements. To accomplish this, it uses a single mechanism—locks—but equips each type of transaction with its own type of lock: *ACID* and *alkaline* locks, which share the name of their respective transactions, and *saline* locks, which are used by BASE transactions.

ACID locks work as in traditional ACID systems. There are ACID locks for both read and write operations; reads

[2]Here the term *transactions* refers to both ACID and BASE transactions, as well as alkaline subtransactions.



(a) *BASE* waits until *ACID* commits. (b) $BASE_2$ waits only for $alkaline_1$... (c) ...but *ACID* must wait all of *BASE* out.

Fig. 3: Examples of concurrent executions of ACID and BASE transactions in Salt.

conflict with writes, while writes conflict with both reads and writes (see the dark-shaded area of Table 2). The duration for which an ACID lock is held depends on the operation type and the chosen isolation level. Operations in $\mathcal{L}$ require *long-term* locks, which are acquired at the start of the operation and are maintained until the end of the transaction. Operations in $\mathcal{S} \setminus \mathcal{L}$ require *short-term* locks, which are only held for the duration of the operation.

Alkaline locks keep alkaline subtransactions isolated from other ACID and alkaline subtransactions. As a result, as Table 2 (light-and-dark shaded subtable) shows, only read-read accesses are considered non-conflicting for any combination of ACID and alkaline locks. Similar to ACID locks, alkaline locks can be either long-term or short-term, depending on the operation type; long-term alkaline locks, however, are only held until the end of the current alkaline subtransaction, and not for the entire duration of the parent BASE transaction: their purpose is solely to isolate the alkaline subtransaction containing the operation that acquired the lock.

Saline locks owe their name to their delicate charge: isolating ACID transactions from BASE transactions, while at the same time allowing for increased concurrency by exposing intermediate states of BASE transactions to other BASE transactions. To that end, (see Table 2) saline locks conflict with ACID locks for non read-read accesses, but *never* conflict with either alkaline or saline locks. Once again, there are long-term and short-term saline locks: short-term saline locks are released after the operation completes, while long-term locks are held until the end of the current BASE transaction. In practice, since alkaline locks supersede saline locks, we acquire only an alkaline lock at the start of the operation and, if the lock is longterm, "downgrade" it at the end of the alkaline subtransaction to a saline lock, to be held until after the end of the BASE transaction.

Figure 3 shows three simple examples that illustrate how ACID and BASE transactions interact. In Figure 3(a), an ACID transaction holds an ACID lock on x, which causes the BASE transaction to wait until the ACID transaction has committed, before it can acquire the lock on x. In Figure 3(b), instead, transaction $BASE_2$ need only wait until the end of $alkaline_1$, before acquiring the lock on x. Finally, Figure 3(c) illustrates the use of saline locks. When $alkaline_1$ commits, it downgrades its lock on x to a saline lock that is kept until the end of the parent BASE transaction, ensuring that the ACID and BASE transactions remain isolated.

Indirect dirty reads In an ACID system the Isolation property holds among any two transactions, making it quite natural to consider only direct interactions between pairs of transactions when defining the undesirable phenomena prevented by the four ANSI isolation levels. In a system that uses Salt isolation, however, the Isolation property covers only *some* pairs of transactions: pairs of BASE transactions are exempt. Losing Isolation's universal coverage has the insidious effect of introducing indirect instances of those undesirable phenomena.

The example in Figure 4 illustrates what can go wrong if Salt Isolation is enforced naively. For concreteness, assume that ACID transactions require a read-committed isolation level. Since Isolation is not enforced between $BASE_1$ and $BASE_2$, $w(y)$ may reflect the value of x that was written by $BASE_1$. Although Isolation is enforced between $ACID_1$ and $BASE_2$, $ACID_1$ ends up reading x's uncommitted value, which violates that transaction's isolation guarantees.

The culprit for such violations is easy to find: dirty reads can indirectly relate two transactions ($BASE_1$ and $ACID_1$ in Figure 4) without generating a direct conflict between them. Fortunately, none of the other three phenomena that ACID isolation levels try to avoid can do the same: for such phenomena to create an indirect relation between two transactions, the transactions at the two ends of the chain must be in direct conflict.

Our task is then simple: we must prevent indirect dirty reads.[3] Salt avoids them by restricting the order in which saline locks are released, in the following two ways:

Read-after-write across transactions A BASE transaction B_r that reads a value x, which has been written by another BASE transaction B_w, cannot release its saline lock on x until B_w has released its own saline lock on x.

Write-after-read within a transaction An operation o_w that writes a value x cannot release its saline

[3] Of course, indirect dirty reads are allowed if ACID transactions require the read-uncommitted isolation level, which does not try to prevent dirty-reads.

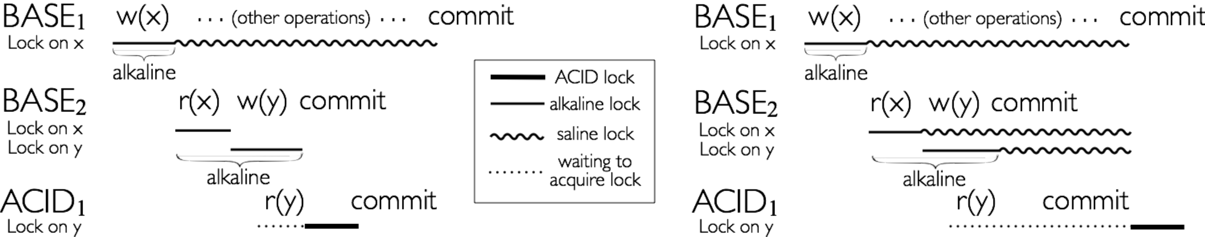


Fig. 4: $ACID_1$ indirectly reads the uncommitted value of x.

Fig. 5: How Salt prevents indirect dirty reads.

lock on x until *all* previous read operations within the same BASE transaction have released their saline locks on their respective objects.[4]

The combination of these two restrictions ensures that, as long as a write remains uncommitted (i.e. its saline lock has not been released) subsequent read operations that observe that written value and subsequent write operations that are affected by that written value will not release their own saline locks. This, in turn, guarantees that an ACID transaction cannot observe an uncommitted write, since saline locks are designed to be mutually exclusive with ACID locks. Figure 5 illustrates how enforcing these two rules prevents the indirect dirty read of Figure 4. Observe that transaction $BASE_2$ cannot release its saline lock on x until $BASE_1$ commits (read-after-write across transactions) and $BASE_2$ cannot release its saline lock on y before releasing its saline lock on x (write-after-read within a transaction).

We can now prove [43] the following Theorem for any system composed of ACID and BASE transactions that enforces Salt Isolation.

Theorem 1. *[Correctness] Given isolation level $\mathcal{A}$, all ACID transactions are protected (both directly and, where applicable, indirectly) from all the undesirable phenomena prevented by $\mathcal{A}$.*

Clarifying serializability The strongest ANSI lock-based isolation level, *locking-serializable* [18], not only prevents the four undesirable phenomena we mentioned earlier, but, in ACID-only systems, also implies the familiar definition of serializability, which requires the outcome of a serializable transaction schedule to be equal to the outcome of a serial execution of those transactions.

This implication, however, holds only if *all* transactions are isolated from all other transactions [18]; this is not desirable in a Salt database, since it would require isolating BASE transactions from each other, impeding Salt's performance goals.

Nonetheless, a Salt database remains true to the essence of the locking-serializable isolation level: it continues to protect its ACID transactions from all four undesirable phenomena, with respect to both BASE transactions and other ACID transactions. In other words, even though the presence of BASE transactions prevents the familiar notion of serializability to "emerge" from universal pairwise locking-serializability, ACID transactions enjoy in Salt the same kind of "perfect isolation" they enjoy in a traditional ACID system.

[4] A write can indirectly depend on any previous read within the same transaction, through the use of transaction-local variables.

6 Implementation

We implemented a Salt prototype by modifying MySQL Cluster [9], a popular distributed database, to support BASE transactions and enforce Salt Isolation. MySQL Cluster follows a standard approach among distributed databases: the database is split into a number of partitions and each partition uses a master-slave protocol to maintain consistency among its replicas, which are organized in a chain. To provide fairness, MySQL Cluster places operations that try to acquire locks on objects in a per-object queue in lock-acquisition order; Salt leverages this mechanism to further ensure that BASE transactions cannot cause ACID transactions to starve.

We modified the locking module of MySQL Cluster to add support for alkaline and saline locks. These modifications include support for (a) managing lock conflicts (see Table 2), (b) controlling when each type of lock should be acquired and released, as well as (c) a queuing mechanism that enforces the order in which saline locks are released, to avoid indirect dirty reads. Our current prototype uses the read-committed isolation level, as it is the only isolation level supported by MySQL Cluster. The rest of this section discusses the implementation choices we made with regard to availability, durability and consistency, as well as an important optimization we implemented in our prototype.

6.1 Early commit for availability

To reduce latency and improve availability, Salt supports *early commit* [44] for BASE transactions: a client that issues a BASE transaction is notified that the transaction has committed when its first alkaline subtransaction commits. To ensure both atomicity and durability despite

failures, Salt logs the logic for the entire BASE transaction before its first transaction commits. If a failure occurs before the BASE transaction has finished executing, the system uses the log to ensure that the entire BASE transaction will be executed eventually.

6.2 Failure recovery

Logging the transaction logic before the first alkaline subtransaction commits has the additional benefit of avoiding the need for managing cascading rollbacks of other committed transactions in the case of failures. Since the committed state of an alkaline subtransaction is exposed to other BASE transactions, rolling back an uncommitted BASE transaction would also require rolling back any BASE transaction that may have observed rolled back state. Instead, early logging allows Salt to roll uncommitted transactions forward.

The recovery protocol has two phases: *redo* and *roll forward*. In the first phase, Salt replays its redo log, which is populated, as in ACID systems, by logging asynchronously to disk every operation after it completes. Salt's redo log differs from an ACID redo log in two ways. First, Salt logs both read and write operations, so that transactions with write operations that depend on previous reads can be rolled forward. Second, Salt replays also operations that belong to partially executed BASE transactions, unlike ACID systems that only replay operations of committed transactions. During this phase, Salt maintains a *context* hash table with all the replayed operations and returned values (if any), to ensure that they are not re-executed during the second phase.

During the second phase of recovery, Salt rolls forward any partially executed BASE transactions. Using the logged transaction logic, Salt regenerates the transaction's query plan and reissues the corresponding operations. Of course, some of those operations may have already been performed during the first phase: the *context* hash table allows Salt to avoid re-executing any of these operations and nonetheless have access to the return values of any read operation among them.

6.3 Transitive dependencies

As we discussed in Section 5.1, Salt needs to monitor transitive dependencies that can cause indirect dirty reads. To minimize bookkeeping, our prototype does not explicitly track such dependencies. Instead it only tracks *direct* dependencies among transactions and uses this information to infer the order in which locks should be released.

As we mentioned earlier, MySQL Cluster maintains a per-object queue of the operations that try to acquire locks on an object. Salt adds for each saline lock a pointer to the most recent non-ACID lock on the queue. Before releasing a saline lock, Salt simply checks whether the pointer points to a held lock—an O(1) operation.

```
1   begin BASE transaction
2       Check whether all items exist. Exit otherwise.
3       Select w_tax into @w_tax from warehouse where w_id = : w_id;
4       begin alkaline−subtransaction
5           Select d_tax into @d_tax, next_order_id into @o_id from
                district where w_id = : w_id and d_id = : d_id;
6           Update district set next_order_id = o_id + 1 where w_id =
                : w_id AND d_id = : d_id;
7       end alkaline−subtransaction
8       Select discount into @discount, last_name into @name, credit
            into @credit where w_id = : w_id and d_id = : d_id and
            c_id = : c_id
9       Insert into orders values (: w_id, : d_id, @o_id, ...);
10      Insert into new_orders values (: w_id, : d_id, o_id);
11      For each ordered item, insert an order line, update stock level, and
            calculate order total
12  end BASE transaction
```

Fig. 6: A Salt implementation of the *new-order* transaction in TPC-C. The lines introduced in Salt are shaded.

6.4 Local transactions

Converting an ACID transaction into a BASE transaction can have significant impact on performance, beyond the increased concurrency achieved by enforcing isolation at a finer granularity. In practice, we find that although most of the performance gains in Salt come from fine-grain isolation, a significant fraction is due to a practical reason that compounds those gains: alkaline subtransactions in Salt tend to be small, often containing a single operation.

Salt's *local-transaction* optimization, inspired by similar optimizations used in BASE storage systems, leverages this observation to significantly decrease the duration that locks are being held in Salt. When an alkaline subtransaction consists of a single operation, each partition replica can locally decide to commit the transaction—and release the corresponding locks—immediately after the operation completes. While in principle a similar optimization could be applied also to single-operation ACID transactions, in practice ACID transactions typically consist of many operations that affect multiple database partitions. Reaching a decision, which is a precondition for lock release, typically takes much longer in such transactions: locks must be kept while each transaction operation is propagated along the entire chain of replicas of each of the partitions touched by the transaction and during the ensuing two-phase commit protocol among the partitions. The savings from this optimization can be substantial: single-operation transactions release their locks about one-to-two orders of magnitude faster than non-optimized transactions.[5] Interestingly, these benefits can extend beyond single operation transactions—it is easy to extend the local-transaction optimization to cover also transactions where all operations touch the same object.

[5] This optimization applies only to ACID and alkaline locks. To enforce isolation between ACID and BASE transactions, saline locks must still be kept until the end of the BASE transaction.

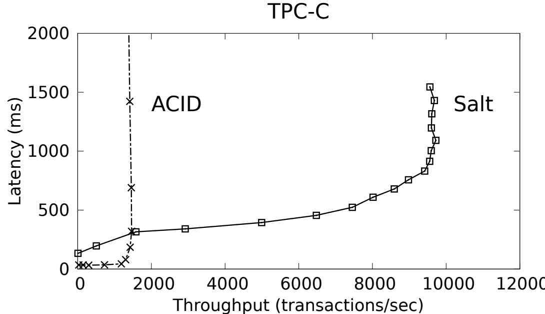


Fig. 7: Performance of ACID and Salt for TPC-C.

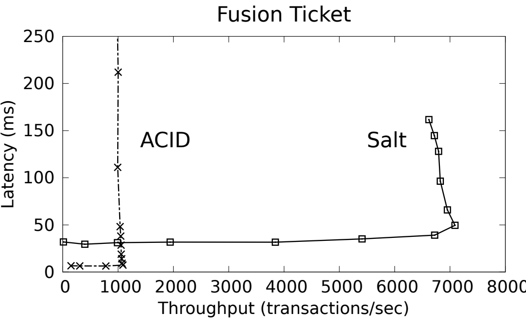


Fig. 8: Performance of ACID and Salt for Fusion Ticket.

7 Case Study: BASE-ifying *new-order*

We started this project to create a distributed database where performance and ease of programming could go hand-in-hand. How close does Salt come to that vision? We will address this question quantitatively in the next section, but some qualitative insight can be gained by looking at an actual example of Salt programming.

Figure 6 shows, in pseudocode, the BASE-ified version of *new-order*, one of the most heavily run transactions in the TPC-C benchmark (more about TPC-C in the next section). We chose *new-order* because, although its logic is simple, it includes all the features that give Salt its edge.

The first thing to note is that BASE-ifying this transaction in Salt required only minimal code modifications (the highlighted lines 2, 4, and 7). The reason, of course, is Salt isolation: the intermediate states of *new-order* are isolated from all ACID transactions, freeing the programmer from having to reason about all possible interleavings. For example, TPC-C also contains the *deliver* transaction, which assumes the following invariant: if an order is placed (lines 9-10), then all order lines must be appropriately filled (line 11). Salt does not require any change to *deliver*, relying on Salt isolation to ensure that *deliver* will never see an intermediate state of *new-order* in which lines 9-10 are executed but line 11 is not.

At the same time, using a finer granularity of isolation between BASE transactions greatly increases concurrency. Consider lines 5-6, for example. They need to be isolated from other instances of *new-order* to guarantee that order ids are unique, but this need for isolation does not extend to the following operations of the transaction. In an ACID system, however, there can be no such distinction; once the operations in lines 5-6 acquire a lock, they cannot release it until the end of the transaction, preventing lines 8-11 from benefiting from concurrent execution.

8 Evaluation

To gain a quantitative understanding of the benefits of Salt with respect to both ease of programming and performance, we applied the ideas of Salt to two applications: the TPC-C benchmark [23] and Fusion Ticket [6].
TPC-C is a popular database benchmark that models online transaction processing. It consists of five types of transactions: *new-order* and *payment* (each responsible for 43.5% of the total number of transactions in TPC-C), as well as *stock-level*, *order-status*, and *delivery* (each accounting for 4.35% of the total).
Fusion Ticket is an open source ticketing solution used by more than 80 companies and organizations [3]. It is written in PHP and uses MySQL as its backend database.

Unlike TPC-C, which focuses mostly on performance and includes only a representative set of transactions, a real application like Fusion Ticket includes several transactions—from frequently used ones such as *create-order* and *payment*, to infrequent administrative transactions such as *publishing* and *deleting-event*—that are critical for providing the required functionality of a fully fledged online ticketing application and, therefore, offers a more accurate view of the programming effort required to BASE-ify entire applications in practice.

Our evaluation tries to answer three questions:

- What is the performance gain of Salt compared to the traditional ACID approach?
- How much programming effort is required to achieve performance comparable to that of a pure BASE implementation?
- How is Salt's performance affected by various workload characteristics, such as contention ratio?

We use TPC-C and Fusion Ticket to address the first two questions. To address the third one, we run a microbenchmark and tune the appropriate workload parameters.

Experimental setup In our experiments, we configure Fusion Ticket with a single event, two categories of tick-

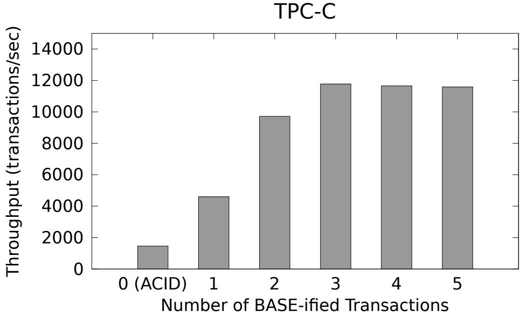


Fig. 9: Incremental BASE-ification of TPC-C.

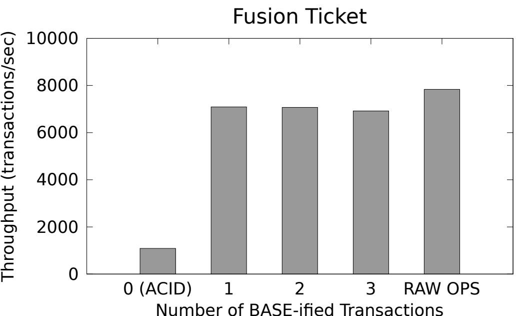


Fig. 10: Incremental BASE-ification of Fusion Ticket.

ets, and 10,000 seats in each category. Our experiments emulate a number of clients that book tickets through the Fusion Ticket application. Our workload consists of the 11 transactions that implement the business logic necessary to book a ticket, including a single administrative transaction, *delete-order*. We do not execute additional administrative transactions, because they are many orders of magnitude less frequent than customer transactions and have no significant effect on performance. Note, however, that executing more administrative transactions would have incurred no additional programming effort, since Salt allows unmodified ACID transactions to safely execute side-by-side the few performance-critical transactions that need to be BASE-ified. In contrast, in a pure BASE system, one would have to BASE-ify *all* transactions, administrative ones included: the additional performance benefits would be minimal, but the programming effort required to guarantee correctness would grow exponentially.

In our TPC-C and Fusion Ticket experiments, data is split across ten partitions and each partition is three-way replicated. Due to resource limitations, our microbenchmark experiments use only two partitions. In addition to the server-side machines, our experiments include enough clients to saturate the system.

All of our experiments are carried out in an Emulab cluster [16, 42] with 62 Dell PowerEdge R710 machines. Each machine is equipped with a 64-bit quad-core Xeon E5530 processor, 12GB of memory, two 7200 RPM local disks, and a Gigabit Ethernet port.

8.1 Performance of Salt

Our first set of experiments aims at comparing the performance gain of Salt to that of a traditional ACID implementation to test our hypothesis that BASE-ifying only a few transactions can yield significant performance gains.

Our methodology for identifying which transactions should be BASE-ified is based on a simple observation: since Salt targets performance bottlenecks caused by contention, transactions that are good targets for BASE-ification are large and highly-contented. To identify suitable candidates, we simply increase the system load and observe which transactions experience a disproportionate increase in latency.

Following this methodology, for the TPC-C benchmark we BASE-ified two transactions: *new-order* and *payment*. As shown in Figure 7, the ACID implementation of TPC-C achieves a peak throughput of 1464 transactions/sec. By BASE-ifying these two transactions, our Salt implementation achieves a throughput of 9721 transactions/sec—6.6*x* higher than the ACID throughput.

For the Fusion Ticket benchmark, we only BASE-ify one transaction, *create-order*. This transaction is the key to the performance of Fusion Ticket, because distinct instances of *create-order* heavily contend with each other. As Figure 8 shows, the ACID implementation of Fusion Ticket achieves a throughput of 1088 transactions/sec, while Salt achieves a throughput of 7090 transactions/sec, 6.5*x* higher than the ACID throughput. By just BASE-ifying *create-order*, Salt can significantly reduce how long locks are held, greatly increasing concurrency.

In both the TPC-C and Fusion Ticket experiments Salt's latency under low load is higher than that of ACID. The reason for this disparity lies in how requests are made durable. The original MySQL Cluster implementation returns to the client *before* the request is logged to disk, providing no durability guarantees. Salt, instead, requires that all BASE transactions be durable before returning to the client, increasing latency. This increase is exacerbated by the fact that we are using MySQL Cluster's logging mechanism, which—having been designed for asynchronous logging—is not optimized for low latency. Of course, this phenomenon only manifests when the system is under low load; as the load increases, Salt's performance benefits quickly materialize: Salt outperforms ACID despite providing durability guarantees.

8.2 Programming effort vs Throughput

While Salt's performance over ACID is encouraging, it is only one piece of the puzzle. We would like to further understand how much programming effort is required to achieve performance comparable to that of a pure BASE

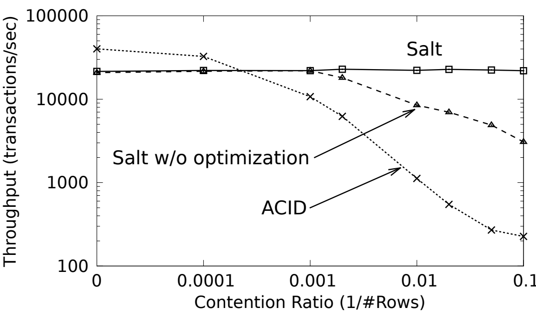


Fig. 11: Effect of contention ratio on throughput.

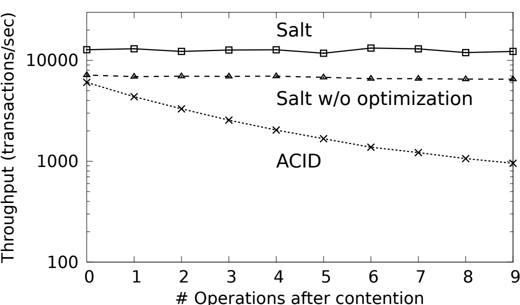


Fig. 12: Effect of contention position on throughput.

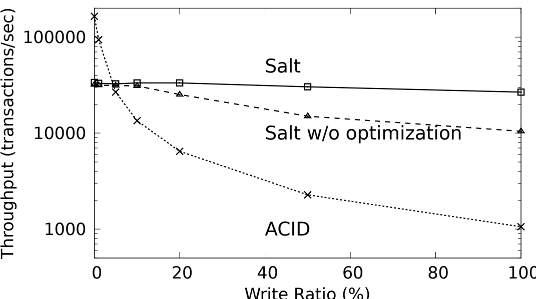


Fig. 13: Effect of read-write ratio on throughput.

implementation—i.e. where all transactions are BASE-ified. To that end, we BASE-ified as many transactions as possible in both the TPC-C and Fusion Ticket codebases, and we measured the performance they achieve as we increase the number of BASE-ified transactions.

Figure 9 shows the result of incrementally BASE-ifying TPC-C. Even with only two BASE-ified transactions, Salt achieves 80% of the maximum throughput of a pure BASE implementation; BASE-ifying three transactions actually reaches that throughput. In other words, there is no reason to BASE-ify the remaining two transactions. In practice, this simplifies a developer's task significantly, since the number of state interleavings to be considered increases exponentially with each additional transactions that need to be BASE-ified. Further, real applications are likely to have proportionally fewer performance-critical transactions than TPC-C, which, being a performance benchmark, is by design packed with them.

To put this expectation to the test, we further experimented with incrementally BASE-ifying the Fusion Ticket application. Figure 10 shows the results of those experiments. BASE-fying one transaction was quite manageable: it took about 15 man-hours—without prior familiarity with the code—and required changing 55 lines of code, out of a total of 180,000. BASE-ifying this first transaction yields a benefit of 6.5*x* over ACID, while BASE-ifying the next one or two transactions with the highest contention does not produce any additional performance benefit.

What if we BASE-ify more transactions? This is where the aforementioned exponential increase in state interleavings caught up with us: BASE-ifying a fourth or fifth transaction appeared already quite hard, and seven more transactions were waiting behind them in the Fusion Ticket codebase! To avoid this complexity and still test our hypothesis, we adopted a different approach: we broke down all 11 transactions into raw operations. The resulting system does not provide, of course, any correctness guarantees, but at least, by enabling the maximum degree of concurrency, it lets us measure the maximum throughput achievable by Fusion Ticket. The result of this experiment is labeled RAW OPS in Figure 10. We find it promising that, even by BASE-ifying only one transaction, Salt is within 10% of the upper bound of what is achievable with a BASE approach.

8.3 Contention

To help us understand how contention affects the performance of Salt, we designed three microbenchmarks to compare Salt, with and without the local-transaction optimization, to an ACID implementation.

In the first microbenchmark, each transaction updates five rows, randomly chosen from a collection of *N* rows. By tuning *N*, we can control the amount of contention in our workload. Our Salt implementation uses BASE transactions that consist of five alkaline subtransactions—one for each update.

Figure 11 shows the result of this experiment. When there is no contention, the throughput of Salt is somewhat lower than that of ACID, because of the additional bookkeeping overhead of Salt (e.g., logging the logic of the entire BASE transaction). As expected, however, the throughput of ACID transactions quickly decreases as the contention ratio increases, since contending transactions cannot execute in parallel. The non-optimized version of Salt suffers from this degradation, too, albeit to a lesser degree; its throughput is up to an order of magnitude higher than that of ACID when the contention ratio is high. The reason for this increase is that BASE transactions contend on alkaline locks, which are only held for the duration of the current alkaline subtransactions and

are thus released faster than ACID locks. The optimized version of Salt achieves further performance improvement by releasing locks immediately after the operation completes, without having to wait for the operation to propagate to all replicas or wait for a distributed commit protocol to complete. This leads to a significant reduction in contention; so much so, that the contention ratio appears to have negligible impact on the performance of Salt.

The goal of the second microbenchmark is to help us understand the effect of the relative position of contending operations within a transaction on the system throughput. This factor can impact performance significantly, as it affects how long the corresponding locks must be held. In this experiment, each transaction updates ten rows, but only one of those updates contends with other transactions by writing to one row, randomly chosen from a collection of ten shared rows. We tune the number of operations that follow the contending operation within the transaction, and measure the effect on the system throughput.

As Figure 12 shows, ACID throughput steadily decreases as the operations that follow the contending operation increase, because ACID holds an exclusive lock until the transaction ends. The throughput of Salt, however, is not affected by the position of contending operations because BASE transactions hold the exclusive locks—alkaline locks—only until the end of the current alkaline subtransaction. Once again, the local-transaction optimization further reduces the contention time for Salt by releasing locks as soon as the operation completes.

The third microbenchmark helps us understand the performance of Salt under various read-write ratios. The read-write ratio affects the system throughput in two ways: (i) increasing writes creates more contention among transactions; and (ii) increasing reads increases the overhead introduced by Salt over traditional ACID systems, since Salt must log read operations, as discussed in Section 6. In this experiment each transaction either reads five rows or writes five rows, randomly chosen from a collection of 100 rows. We tune the percentage of read-only transactions and measure the effect on the system throughput.

As Figure 13 shows, the throughput of ACID decreases quickly as the fraction of writes increases. This is expected: write-heavy workloads incur a lot of contention, and when transactions hold exclusive locks for long periods of time, concurrency is drastically reduced. The performance of Salt, instead, is only mildly affected by such contention, as its exclusive locks are held for much shorter intervals. It is worth noting that, despite Salt's overhead of logging read operations, Salt outperforms ACID even when 95% of the transactions are read-only transactions.

In summary, our evaluation suggests that, by holding locks for shorter times, Salt can reduce contention and offer significant performance improvements over a traditional ACID approach, without compromising the isolation guarantees of ACID transactions.

9 Related Work

ACID Traditional databases rely on ACID's strong guarantees to greatly simplify the development of applications [7–9, 12–14, 22, 36]. As we noted, however, these guarantees come with severe performance limitations, because of both the coarse granularity of lock acquisitions and the need for performing a two-phase commit (2PC) protocol at commit time.

Several approaches have been proposed to improve the performance of distributed ACID transactions by eliminating 2PC whenever possible. H-Store [39], Granola [24], and F1 [38] make the observation that 2PC can be avoided for transactions with certain properties (e.g. partition-local transactions). Sagas [29] and Lynx [44] remark that certain large transactions can be broken down into smaller ones without affecting application semantics. Lynx uses static analysis to identify eligible transactions automatically. Our experience with TPC-C and Fusion Ticket, however, suggests that performance critical transactions are typically complex, making them unlikely to be eligible for such optimizations. Calvin [40] avoids using 2PC by predefining the order in which transactions should execute at each partition. To determine this order, however, one must be able to predict which partitions a transaction will access before the transaction is executed, which is very difficult for complex transactions. Additionally, using a predefined order prevents the entire system from making progress when any partition becomes unavailable.

BASE To achieve higher performance and availability, many recent systems have adopted the BASE approach [1, 15, 20, 21, 27, 34]. These systems offer a limited form of transactions that only access a single item.

To mitigate somewhat the complexity of programming in BASE, several solutions have been proposed to provide stronger semantics. ElasTraS [25], Megastore [17], G-Store [26], and Microsoft's Cloud SQL Server [19] provide ACID guarantees within a single partition or key group. G-Store and ElasTraS further allow dynamic modification of such key groups. These systems, however, offer no atomicity or isolation guarantees across partitions. Megastore further provides the option of using ACID transactions, but in an all-or-nothing manner: either all transactions are ACID or none of them are.

10 Conclusion

The ACID/BASE dualism has to date forced developers to choose between ease of programming and performance. Salt shows that this choice is a false one. Using the new abstraction of BASE transactions and a mechanism to properly isolate them from their ACID counterparts, Salt enables for the first time a tenable middle ground between the ACID and BASE paradigms; a middle ground where performance can be incrementally attained by gradually increasing the programming effort required. Our experience applying Salt to real applications matches the time-tested intuition of Pareto's law: a modest effort is usually enough to yield a significant performance benefit, offering a drama-free path to growth for companies whose business depends on transactional applications.

Acknowledgements

Many thanks to our shepherd Willy Zwaenepoel and to the anonymous reviewers for their insightful comments. Lidong Zhou, Mike Dahlin, and Keith Marzullo provided invaluable feedback on early drafts of this paper, which would not have happened without the patience and support of the Utah Emulab team throughout our experimental evaluation. This material is based in part upon work supported by a Google Faculty Research Award and by the National Science Foundation under Grant Number CNS-1409555. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Apache HBase. http://hbase.apache.org/.

[2] AuctionMark. http://hstore.cs.brown.edu/projects/auctionmark/.

[3] Current users of Fusion Ticket. http://www.fusionticket.com/hosting/our-customers.

[4] Dolibarr. http://www.dolibarr.org/.

[5] E-venement. http://www.e-venement.org/.

[6] Fusion Ticket. http://www.fusionticket.org.

[7] MemSQL. http://www.memsql.com/.

[8] Microsoft SQL Server. http://www.microsoft.com/sqlserver/.

[9] MySQL Cluster. http://www.mysql.com/products/cluster/.

[10] Ofbiz. http://ofbiz.apache.org/.

[11] Openbravo. http://www.openbravo.com/.

[12] Oracle Database. http://www.oracle.com/database/.

[13] Postgres SQL. http://www.postgresql.org/.

[14] SAP Hana. http://www.saphana.com/.

[15] SimpleDB. http://aws.amazon.com/simpledb/.

[16] Utah Emulab. http://www.emulab.net/.

[17] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[18] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[19] Philip A Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for Cloud Computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1255–1263. IEEE, 2011.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, Berkeley, CA, USA, 2006. USENIX Association.

[21] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[22] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[23] Transaction Processing Performance Council. TPC benchmark C, Standard Specification Version 5.11, 2010.

[24] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012. USENIX.

[25] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: an elastic transactional data store in the cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[26] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[28] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 78–91, New York, NY, USA, 1997. ACM.

[29] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987.

[30] Seth Gilbert and Nancy Ann Lynch. Perspectives on the CAP Theorem. Institute of Electrical and Electronics Engineers, 2012.

[31] James N Gray. *Notes on data base operating systems*. Springer, 1978.

[32] Pat Helland. Life beyond Distributed Transactions: an Apostate's

Opinion. In *Third Biennial Conference on Innovative Data Systems Research*, pages 132–141, 2007.

[33] George Roy Hill and William Goldman. Butch Cassidy and the Sundance Kid. Clip at `https://www.youtube.com/watch?v=1IbStIb9XXw`, October 1969.

[34] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44:35–40, April 2010.

[35] Michael McLure. Vilfredo Pareto, 1906 Manuale di Economia Politica, Edizione Critica, Aldo Montesano, Alberto Zanni and Luigino Bruni (eds). *Journal of the History of Economic Thought*, 30(01):137–140, 2008.

[36] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[37] Dan Pritchett. BASE: An Acid Alternative. *Queue*, 6:48–55, May 2008.

[38] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littleeld, and Phoenix Tong. F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778. ACM, 2012.

[39] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[40] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[41] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[42] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[43] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a Distributed Database (extended version). Technical Report TR-14-10, Department of Computer Science, The University of Texas at Austin, September 2014.

[44] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.

Phase Reconciliation for Contended In-Memory Transactions

Neha Narula, Cody Cutler, Eddie Kohler[†], and Robert Morris
MIT CSAIL and [†]Harvard University

Abstract

Multicore main-memory database performance can collapse when many transactions contend on the same data. Contending transactions are executed serially—either by locks or by optimistic concurrency control aborts—in order to ensure that they have serializable effects. This leaves many cores idle and performance poor. We introduce a new concurrency control technique, *phase reconciliation*, that solves this problem for many important workloads. Doppel, our phase reconciliation database, repeatedly cycles through *joined*, *split*, and *reconciliation phases*. Joined phases use traditional concurrency control and allow any transaction to execute. When workload contention causes unnecessary serial execution, Doppel switches to a split phase. There, updates to contended items modify *per-core* state, and thus proceed in parallel on different cores. Not all transactions can execute in a split phase; for example, all modifications to a contended item must commute. A reconciliation phase merges these per-core states into the global store, producing a complete database ready for joined-phase transactions. A key aspect of this design is determining which items to split, and which operations to allow on split items.

Phase reconciliation helps most when there are many updates to a few popular database records. Its throughput is up to 38× higher than conventional concurrency control protocols on microbenchmarks, and up to 3× on a larger application, at the cost of increased latency for some transactions.

1 Introduction

The key to good multicore performance and scalability is the elimination of serial execution. Cores should make progress in parallel whenever possible; the implementation should not force cores to wait for one another.

But serial execution sometimes appears to be an inherent feature of a problem. Most databases, for example, guarantee *serializable* results: the effect of executing a set of transactions in parallel should equal the effect of the same transactions executed in some serial order. This requires care when concurrent transactions conflict, which happens when one of them writes a record that the other either reads or writes. Database concurrency control protocols—mostly variants of two-phase locking (2PL) or optimistic concurrency control (OCC)—enforce serializability on conflicting transactions by executing them serially: one transaction will wait for the other, either by spinning on a lock (2PL) or by aborting and retrying (OCC).

Unfortunately, conflicts are common in some important real-world database workloads. For instance, consider an auction web site with skewed item popularity. As a popular item's auction time approaches, and users strive to win the auction, a large fraction of concurrent transactions might update the item's current highest bid. Modern multicore databases will execute these transactions serially, causing huge reductions in throughput.

We present *phase reconciliation*, a new concurrency control technique that can execute some highly conflicting workloads efficiently in parallel, while still guaranteeing serializability; and *Doppel*, a new in-memory database based on phase reconciliation.

Our basic technique is to split logical values across cores. We were inspired by efficient multicore counter designs, such as for packet counters, which partition a logical value into n counters, one per core. To increment the logical counter, a core updates its per-core value; to read it, a core *reconciles* these per-core values into one correct value by adding them together. This design is less contentious than a single global counter as long as writes greatly outnumber reads. But simple value splitting is too restrictive for general database use; splitting every item in the database would explode transaction overhead, and reconciling values on every read is costly. Instead, we dynamically shift data between split and reconciled states, based on observed contention.

A key design decision was to amortize the impact of value reconciliation over many transactions by executing different transactions in different *phases*. In joined phases, the database's structures are accessed using OCC. There are no per-core values and any transaction can execute (albeit with potentially high contention). In split phases, in contrast, updates are applied when possible to split per-core values rather than the global store. This greatly reduces contention on split data, but for correctness not all transactions may execute. Inappropriate uses of split data cause a transaction to block. Finally, short reconciliation phases reconcile these per-core values into the global store. When a reconciliation phase ends, blocked transactions resume and the next joined

phase begins. Thus, conflicting writes operate efficiently in the split phase, reads of frequently-updated data operate efficiently in the joined phase, and the system can achieve high overall performance even for challenging conflicting workloads.

The workloads that work best with phase reconciliation are ones with frequently-updated data items where contentious updates are commutative (they have the same overall effect regardless of order). Commutativity allows different cores to update their per-core values without coordination. Applicable situations include maintenance of the highest bids in the auction example, counts of votes on popular items, and maintenance of "top-*k*" lists for news aggregators such as Reddit [2].

The contributions of this work are the phase reconciliation technique and an implementation of phase reconciliation in Doppel. We show that phase reconciliation improves the overall throughput of various contentious workloads by up to 38× over OCC and 19× over 2PL, and has read throughput comparable to OCC. We port an auction website, RUBiS, to use Doppel and show Doppel improves bidding throughput with popular auctions by up to 3× over OCC.

2 Related Work

The idea of phase reconciliation is related to ideas in transactional memory, executing fast transactions on in-memory databases, and exploiting commutativity to reconcile divergent values, particularly in multicore operating systems and distributed systems.

Transactional memory. In designing Doppel we were inspired by some novel uses of transactional memory. Several designs have been proposed dividing transactions into phases, or rescheduling transactions to avoid aborts. Lev et al. propose the idea of using phases to support executing transactions both on best-effort hardware transactional memory and software transactional memory [21]. We leverage a similar idea to run transactions in different modes which are optimized for the types of transactions in those modes. Sync-Phase splits transactions up into computation and commit phases [25]. We do not split a transaction across phases, but assign transactions to different phases, based on the type of data they access and the operations they perform.

Transactional memory has been used directly for database transactions [20]. These transactions are often too large to use hardware transactional memory in a straightforward manner, so this work develops techniques to split transactions and apply them using timestamp ordering [8]. Still, spurious aborts are common in TM implementations of databases, since some memory writes to index data structures (which abort TM transactions) are irrelevant to database conflicts. One technique for addressing this problem on multicore architectures is rescheduling conflicting operations after detection to avoid continuous retries [7]. On contentious workloads with many conflicting writes, transactional memory would still be forced to abort or run the transactions one at a time. Our techniques would help in this situation.

Main-memory database concurrency control. Conventional wisdom is that when requests in the workload frequently conflict, they must serialize for correctness [16]. Given that, most related work has focused on improving scalability in the database engine for workloads which do not inherently conflict. Several databases try to leverage multiple cores by partitioning the data and running one partition per core. Systems like H-store/VoltDB [28, 29], HyPer [17], and Dora [23] all employ this technique. It is reasonable when the data is perfectly partitionable, but the overhead of cross-partition transactions in these systems is significant, and finding a good partitioning can be difficult. In our problem space (data contention) partitioning won't necessarily help; a single popular record with many writes wouldn't be able to utilize multiple cores. Hyder [9] uses a technique called meld [10], which lets individual servers or cores operate on a snapshot of the database and submit requests for commits to a central log. Each server processes the log and determines commit or abort decisions deterministically. Doppel also processes on local data copies but by restricting transaction execution to phases, can commit without global communication.

Multimed [24] also replicates data per core, but does so for read availability instead of write performance as in Doppel. The central write manager in Multimed is a bottleneck. Doppel partitions *local copies* of data amongst cores for writes and provides a way to re-merge the data for access by other cores.

Doppel uses optimistic concurrency control, of which there have been many variants [4, 8, 10, 18, 19, 30]. We use the algorithm in Silo [30], which is very effective at reducing contention in the commit protocol, but does not reduce contention caused by conflicting data writes. Larson et al. [19] explore optimistic and pessimistic multiversion concurrency control algorithms for main-memory databases, and this work is implemented in Microsoft's Hekaton [14]. This work presents ideas to eliminate contention due to locking and latches; we go further to address the problem of contention caused by conflicting writes to data. In future work we would like to implement a version of Doppel using pessimistic concurrency control. Doppel's split phase techniques are related to ideas which take advantage of commutativity and abstract data types in concurrency control [15, 31].

Multicore scalability. Linux developers have put a lot of effort into achieving parallel performance on multiprocessor systems. Doppel adopts ideas from the multicore scalability community, including the use of

commutativity to remove scalability bottlenecks [13]. OpLog [11] uses the idea of per-core data structures on contentious write workloads to increase parallelism, and Refcache [12] uses per-core counters, deltas, and epochs. This work tends to shift the performance burden from writes onto reads, which reconcile the per-core data structures whenever they execute. Doppel also shifts the burden onto reads, but phase reconciliation aims to reduce this performance burden in absolute terms by amortizing the effect of reconciliation over many transactions. Our contribution is making these ideas work in a larger transaction system.

Distributed consistency. Some work in distributed systems has explored the idea of using commutativity to reduce concurrency control, usually forgoing serializability. RedBlue consistency [22] uses the idea of blue, eventually consistent local operations which do not require coordination and red, consistent operations which do. Blue phase operations are analagous to Doppel's operations in the split phase. Walter [27] uses the idea of counting sets to avoid conflicts. Doppel could use any Conflict-Free Replicated Data Type (CRDT) [26] with its update operations in the split phase, but does not limit data items to specific operations outside the split phase.

One way of thinking about phase reconciliation is that by restricting operations only *during* phases but not *between* them, we support both scalable (per-core) implementations of commutative operations and efficient implementations of non-commutative operations on the same data items.

3 System model

We implemented phase reconciliation in a multicore, in-memory database called Doppel. Doppel has a low-level key/value store interface, and clients submit transactions in the form of procedures. Doppel provides serializable transactions.

Doppel transactions are *one-shot*: once begun, a transaction runs to completion without communication or disk I/O. Combined with an in-memory database, this means threads will not block due to user or disk stalls. One-shot transactions are used extensively in online transaction processing workloads [5, 28]. *Worker* threads, one per core, run transactions.

Our implementation of Doppel does not currently provide durability. Existing work suggests that asynchronous batched logging could be added to Doppel without becoming a bottleneck [19, 30].

Doppel records have typed values, and each type supports one or more operations. Transactions interact with the database via calls to operations. For example, the MAX(k,n) operation looks up an integer record with key k, and sets its value to the maximum of its current value and n. Some operations return values—GET(k), for example, returns the value of key k—and others do not; some operations modify the database and others do not. Each operation accesses exactly one database record. This isn't a functional restriction: users can build multi-record operations from single-record ones using transactions.

4 Split operations

A phase reconciliation database, such as Doppel, detects contended database records and, during split and reconciliation phases, marks them as *split*. For such records, operations that would normally contend can proceed in parallel.

1. At the beginning of each split phase, Doppel initializes *per-core slices* for each split record. There is one slice per contended record per core.

2. During the split phase, all operations on split records are applied to their per-core slices.

3. During the reconciliation phase, the per-core slices are merged back into the global store.

The combination of applying the operation to a slice and the merge step should have the same effect as the operation would normally. However, the code required to update a slice may be quite different from the code required to update a normal record.

To ensure good performance, per-core slices must be quick to initialize, and operations on slices must be fast. Most critically, the merging step, where per-core slices are merged into the global store, must take $O(J)$ time where J is the number of cores, instead of $O(N)$ time where N is the number of operations applied. This precludes some designs. For instance, one might think that split-phase execution could log updates to per-core slices, with the reconciliation phase applying the logged updates in time order; but this would cause those updates to execute serially, exactly the performance problem we want to avoid.

To ensure correctness, Doppel must ensure serializability. Executing transactions concurrently in a split phase must have the same effects as executing those same transactions in some serial order. Specifically, consider the set of transactions that commit in some split phase. Then there must exist a serial order of those transactions that satisfies:

1. The result of merging per-core slices with the global store is the same as if the transactions had executed, in the serial order, against the global store.

2. Every operation executed on a split record gets the same return value as if it had executed, in the serial order, against the global store.

3. Every operation executed on the global store gets the same return value as it would in the serial order.

An example of an operation that meets these requirements is MAX(k,n) on integer records, which assigns $v[k] \leftarrow \max\{v[k],n\}$ and returns nothing. When Doppel detects contention on MAX(k,n) operations for some key k, it marks k as split for MAX. When entering the next split phase, Doppel initializes per-core slices $c_j[k]$ with the global value $v[k]$. When a transaction on core j commits an operation MAX(k,n), Doppel sets $c_j[k] \leftarrow \max\{c_j[k],n\}$. Key k is temporarily reserved for MAX operations; a transaction that tries to execute another kind of operation on k will block until the following joined phase. When the split phase is over, Doppel merges the per-core slices by setting $v[k] \leftarrow \max_j c_j[k]$.

This implementation of MAX is efficient because per-core slices are fast to initialize, fast to update, and fast to merge. If many concurrent transactions call MAX(k,n) during a split phase, Doppel executes them in parallel on multiple cores with no coordination, getting good parallel speedup over the serial execution of conventional OCC or locking. Another reason for efficiency is that Doppel avoids expensive cache line transfers relating to contended data; these can make OCC and locking on many cores slower than serial execution on a single core.

Doppel's implementation is also correct. The main reason is that MAX *commutes* with itself: the effect of a set of MAX(k,n) operations on $v[k]$ is independent of their order. When operations do not commute, Doppel must enforce a serial order on those operations using global coordination. Per-core slices, which avoid coordination by design, thus work only for commutative operations. It's also important that Doppel restricts key k during the split phase to accept MAX operations only. This means that *all* split-phase operations on k commute, and it's safe to apply them to the per-core slices (even though the slices suppress information about the global execution order). Finally, MAX returns nothing, which is trivially the same as it would return when executed against the global store. We extend this argument in §5.6.

Doppel supports several splittable operations beyond MAX. We ensure these operations are both fast and correct by following some simple guidelines; a more complex implementation could relax these guidelines somewhat, as long as it still achieved the properties above.

1. Every splittable operation must commute with itself.

2. Every splittable operation must return nothing.

3. The system selects one splittable operation per split record per split phase. The selected operation can change between phases—for example, the operation for key k might be MIN in one split phase, and MAX in the next—but within a given phase, any operation but the selected operation causes the containing transaction to abort (and retry in the next joined phase).

4. The size of a per-core slice is independent of the number of operations that executed on that slice.

Doppel's current splittable operations are as follows.

- MAX(k,n) and MIN(k,n) replace k's integer value with the maximum/minimum of it and n.

- ADD(k,n) adds n to k's integer value.

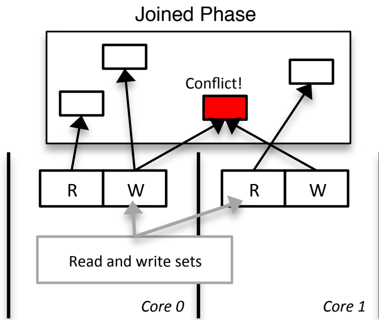
- OPUT(k,o,x) is an operation on *ordered tuples*. An ordered tuple is a 3-tuple (o,j,x) where o, the *order*, is a number (or several numbers in lexicographic order); j is the ID of the core that wrote the tuple; and x is an arbitrary byte string. If k's current value is (o,j,x) and OPUT(k,o',x') is executed by core j', then k's value is replaced by (o',j',x') if $o' > o$, or if $o' = o$ and $j' > j$. Absent records are treated as having $o = -\infty$. The order and core ID components make OPUT commutative. Doppel also supports the usual PUT(k,x) operation for any type, but this doesn't commute and thus cannot be split.

- TOPKINSERT(k,o,x) is an operation on *top-K sets*. A top-K set is like a bounded set of ordered tuples: it contains at most K items, where each item is a 3-tuple (o,j,x) of order, core ID, and byte string. When core j' executes TOPKINSERT(k,o',x'), Doppel inserts the tuple (o',j',x') into the relevant top-K set. At most one tuple per order value is allowed: in case of duplicate order, the record with the highest core ID is chosen. If the top-K contains more than K tuples, the system then drops the tuple with the smallest order. Again, the order and core ID components make TOPKINSERT commutative.

More operations could easily be added (for instance, multiply).

5 Design

This section describes phase reconciliation in the context of Doppel. First, we describe the three phases of phase reconciliation. Second, we describe how updates are reconciled and how records are marked as either *split* or *reconciled*. Next, we describe how the system transitions between phases. We close with a brief argument that Doppel's implementation produces serializable results.



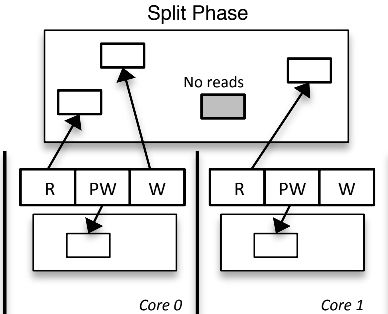


Figure 1: Concurrent transactions executing on different cores, shown in the joined phase and split phase. In the split phase certain data is split so that writes don't conflict.

```
Data: read set R, write set W
// Part 1
for record, operation in sorted(W) do
    lock(record);
commit-tid ← generate-tid()
// Part 2
for record, read-tid in R do
    if record.tid ≠ read-tid
        or (record.locked and record ∉ W)
    then abort();
// Part 3
for record, operation in W do
    apply(operation, record, commit-tid);
    unlock(record);
```

Figure 2: Doppel's joined phase commit protocol. Fences are elided.

5.1 Joined phase execution

A joined phase can execute any transaction. All records are reconciled—there is no notion of split data and there are no per-core slices—so the protocol treats all records the same.

Joined-phase execution could use any concurrency control protocol. However, some designs make more sense for overall performance than others. If all is working according to plan, the joined phase will have few conflicts; transactions that conflict should execute in the split phase. This is why Doppel's joined phase uses optimistic concurrency control (OCC), which performs better than locking when conflicts are rare.

The left side of Figure 1 shows two transactions executing on different cores in a joined phase, and Figure 2 shows the joined-phase commit protocol, which is based on that of Silo [30]. Records have transaction IDs (TIDs); these indicate the ID of the last transaction to write the non-split record, and help detect conflicts. A read set and a write set are maintained for each executing transaction. During execution, a transaction buffers its writes and records the TIDs for all values read or written in its read set. At commit time, the transaction locks the records in its write set (in a global order to prevent deadlock) and aborts if any are locked; obtains a TID; validates its read set, aborting if any values in the read set have changed since they were read, or are concurrently locked by other transactions; and finally writes the new values and TIDs to the shared store.

To avoid overhead and contention on TID assignment, our implementation assigns TIDs locally, using per-core information and the TIDs in the read set. The resulting commit protocol is serializable, but the TID order might diverge from the serial order.

Each transaction executes within a single phase. Any transaction that commits in a joined phase executed completely within that joined phase. Doppel thus cannot leave a joined phase for the following split phase until all current transactions commit or abort. As we see below, this requires coordination across threads.

5.2 Split phase execution

A split phase can execute in parallel some transactions that would normally contend. Accesses to reconciled data proceed much as in a joined phase, using OCC, but split-data operations execute on per-core slices. Split phases cannot execute all transactions, however. As we saw in §4, Doppel selects one operation per split record per split phase. A transaction that invokes an unselected operation on a split record will be aborted and *stashed* for restart during the next joined phase.

The right side of Figure 1 shows a split phase, with each transaction writing to per-core slices. For example, a transaction that executed an ADD$(k, 10)$ operation on a split numeric record might add 10 to the local core's slice for that record.

When a split phase transaction commits, Doppel uses the algorithm in Figure 3. It is similar to the algorithm in Figure 2 with a few important differences. The write set W contains only un-split data, while SW buffers updates to split data. The commit protocol applies the SW updates to the per-core slices. Since these slices are inherently

```
Data: read set R, reconciled write set W, split
      write set SW
// Part 1
for record, operation in sorted(W) do
    lock(record);
commit-tid ← generate-tid()
// Part 2
for record, read-tid in R do
    if record.tid ≠ read-tid
        or (record.locked and record ∉ W)
    then abort();
// Part 3
for record, operation in W do
    apply(operation, record, commit-tid);
    unlock(record);
for slice, operation in SW do
    slice-apply(operation, slice, commit-tid);
```

Figure 3: Doppel's split phase commit protocol.

```
Data: per-core slices S for core j
for record, operation, slice in S do
    lock(record);
    merge-apply(operation, slice, record);
    unlock(record);
S ← ∅
```

Figure 4: Doppel's per-core reconciliation phase protocol.

invisible to concurrently running transactions, there is no need to lock them or check their version numbers. (Any concurrent transaction must be running on another core, since each core runs transactions to completion one at a time.)

Any transaction that commits in a split phase executed completely within that split phase; Doppel does not enter the following joined phase until all of the split-phase transactions commit or abort.

5.3 Reconciliation phase execution

During a reconciliation phase, each core stops processing transactions and merges its per-core slices with the global store. For example, for a split record that used Max, each core locks the global record, sets its value to the maximum of the previous value and its per-core slice, and unlocks the record. This involves serial processing of the per-core slices, but the expense is amortized over all the transactions that executed in the split phase. The per-core slices are then cleared and the database enters the next joined phase.

5.4 Phase transitions

Transitions between phases are managed by a coordinator thread and apply globally, across the entire database. To initiate a transition from a joined phase to the next split phase, the coordinator begins by publishing the phase change in a global variable. Workers check this variable between transactions; when they notice a change, they stop processing new transactions, acknowledge the change, and wait for permission to proceed. When all workers have acknowledged the change, the coordinator releases them, and workers start executing transactions in split mode. A similar process transitions from a split phase to the next reconciliation phase. When a split-phase worker notices a transition to the reconciliation phase, it stops processing transactions, merges its per-core slices with the global store, and then acknowledges the phase transition and waits for permission to proceed. Once all workers have acknowledged the change, the coordinator releases them to the next joined phase; each worker restarts any transactions it stashed in the split phase and starts accepting new transactions. It is safe for reconciliation to proceed in parallel with other cores' split-phase transactions since reconciliation modifies the global versions of split records, while split-phase transactions access per-core slices. No transactions will start joined phase operations on formerly split data until the coordinator has received acknowledgements from all workers for the phase transition, meaning they all finished their merge.

The Doppel coordinator usually starts a phase change every 20 milliseconds, but feedback mechanisms allow it to flexibly adjust to the workload. If, in a joined phase, no records appear contended—or they contend on unsplittable operations—the coordinator delays the next split phase. A worker can also delay a split phase by refusing to acknowledge it, and our workers delay acknowledging a split phase until they have committed or aborted all previously-stashed transactions. Finally, if, in a split phase, workers have to abort and stash too many transactions, the coordinator hurries the next joined phase.

5.5 Classification

Doppel automatically decides how records should be split. During joined execution, Doppel samples transactions' conflicting record accesses, and keeps a count of which records are most conflicted (are causing the most aborts) and by which operations. During the transition to the split phase, a coordinator thread examines these counts and marks the most conflicted records as split data for the next phase. Each core reads this list before the start of the next split phase in order to know which records are restricted. Doppel also samples which transactions are stashed due to incompatible operations on split data during the split phase, and uses this to consider whether to move a split record back to reconciled or change its assigned operation. Since split records in the split phase will not cause conficts, Doppel uses write sampling to estimate if a split record might still be con-

tended.

Doppel also supports manual data labeling ("this record should be split for this operation"), but we only use automatic detection in our experiments.

5.6 Serializability

This section sketches an argument that Doppel transactions are serializable.

Since transactions don't cross phases—any committed transaction executes entirely within a single phase—we can consider phases as units. Joined phases are clearly serializable since they just implement OCC, but to show that split and reconciliation phases are serializable, we must consider per-core slices. So consider a split–reconciliation phase pair that commits a set of transactions. We will show that there is a serial execution of those transactions against the global store—without using per-core slices—that produces the same output global store and the same operation results as the concurrent execution. Since the operations produce identical results, any conditional logic inside the transactions will make identical decisions in concurrent execution as in the serial order, so the transactions as a whole will behave identically.

Consider the transactions that commit in a split phase. These transactions can access both split records and non-split records. The non-split records use OCC, so the transactions are serializable with respect to non-split records. It remains to be shown that at least one serial order valid for non-split records is valid for split records as well. We show that, in fact, *any* serial order that works for non-split records also works for split records. Consider a split record *r* with currently selected operation OP. (We can consider one record at a time because each operation affects only one record.) Since it is splittable, OP commutes with itself and returns nothing. All committed split-phase operations on *r* must use OP, since Doppel aborts transactions that use non-selected operations. So these operations trivially return the same results in any serial order as in the concurrent execution: OP always returns nothing! Commutativity shows that the final value produced by applying the OP operations to the global store is the same regardless of the serial order chosen. This value also equals the outcome of applying the OPS to per-core slices and then merging those slices into the global store, though the reasons why depend on the operation. This concludes the argument.

6 Implementation

Doppel is implemented as a multithreaded server written in Go. Go made thread management and RPC easy, but caused problems with scaling to many cores, particularity in the Go runtime's scheduling and memory management. In our experiments we carefully managed memory allocation to avoid this contention at high core counts.

```
func max-merge(key Key) {
  val := local-get(key)
  g-val := global-get(key)
  global-set(key, max(g-val, val))
}

func oput-merge(key Key,
    phase TID) {
  order, coreid, val := local-get(key)
  // note that coreid == system.MyCoreID()
  g-order, g-coreid, g-val := global-get(key)
  if order > g-order ||
      (order == g-order && coreid > g-coreid) {
    global-set(key, (order, coreid, val))
  }
}
```

Figure 5: Doppel MAX and OPUT merge functions.

Doppel runs one worker thread per core, and one coordinator thread which is responsible for changing phases and synchronizing workers when progressing to a new phase. Doppel uses channels to synchronize phase changes and acknowledgements between the coordinator and workers. It briefly pauses processing transactions while moving between phases; we found that this affected throughput at high core counts. Another design could execute transactions that do not read or write past or future split data while the system is transitioning phases.

Workers read and write to a shared store, which is a set of key/value maps, using per-key locks. The maps are implemented as hash tables. Clients submit transactions written in Go to any worker, indicating the transaction to execute along with arguments. Doppel supports RPC from remote clients over TCP, but we do not measure this in §8. All workers have per-core slices for the split phases.

Developers write transactions in Go with no knowledge of reconciled data, split data, per-core slices, or phases. They access data using a key/value get and set interface or using the operations mentioned in §4.

7 Application Experience

We implemented two test applications: a feature of a social networking site where users can like pages, and a version of the RUBiS auction site benchmark.

The LIKE application simulates a set of users "liking" profile pages. Each update transaction writes a record inserting the user's like of a page, and then increments a per-page sum of likes. Each read transaction reads the user's last like and reads the total number of likes for some page. With a high level of skew, this application

```
func StoreBid(bidder, item, amt) (*Bid, TID) {
  bidkey := NewKey()
  bid := Bid {
    Item: item,
    Bidder: bidder,
    Price: amt,
  }
  Put(bidkey, bid)
  highest := Get(MaxBidKey(item))
  if amt > highest {
    Put(MaxBidKey(item), amt)
    Put(MaxBidderKey(item), bidder)
  }
  numBids := Get(NumBidsKey(item))
  Put(NumBidsKey(item), numBids+1)
  tid := Commit()  // applies writes or aborts
  return &bid, tid
}
```

Figure 6: Original RUBiS `StoreBid` transaction.

```
func StoreBid(bidder, item, amt) (&Bid, TID) {
  bidkey := NewKey()
  bid := Bid {
    Item: item,
    Bidder: bidder,
    Price: amt,
  }
  Put(bidkey, bid)
  Max(MaxBidKey(item), amt)
  OPut(MaxBidderKey(item),
      ([amt, GetTimestamp()], MyCoreID(), bidder))
  Add(NumBidsKey(item), 1)
  TopKInsert(BidsPerItemIndexKey(item),
             amt, bidkey)
  tid := Commit()  // applies writes or aborts
  return &bid, tid
}
```

Figure 7: Doppel `StoreBid` transaction.

explores the case where there are many users but only a few popular pages; thus the increments often conflict, but the inserts of individual records recording user likes do not. We expect the per-page sums for the popular page records to be marked as split data in the split phase, for use with the ADD operation.

We used RUBiS [6], an auction website modeled after eBay, to evaluate Doppel on a realistic application. RUBiS users can register items for auction, place bids, make comments, and browse listings. RUBiS has 7 tables (users, items, categories, regions, bids, buy_now, and comments) and 26 interactions based on 17 database transactions. We ported a RUBiS implementation to Go for use with Doppel.

There are a few notable transactions in the RUBiS workload for which Doppel is particularly suited: `StoreBid`, which inserts a user's bid and updates auction metadata for an item, and `StoreComment`, which publishes a user's comment on an item and updates the rating for the auction owner. RUBiS materializes the `maxBid`, `maxBidder`, and `numBids` per auction, and a `userRating` per user based on comments on an owning user's auction items. We show RUBiS's `StoreBid` transaction in Figure 6.

If an auction is very popular, there is a greater chance two users are bidding or commenting on it at the same time, and that their transactions will issue conflicting writes. At first glance it might not seem like Doppel could help with the `StoreBid` transaction; the auction metadata is contended and could potentially be split, but each `StoreBid` transaction requires reading the current bid to see if it should be updated, and reading the current number of bids to add one. Recall that split data cannot be read during a split phase, so as written in Figure 6 the transaction would have to execute in a joined phase, and would not benefit from local per-core operations.

But note that the `StoreBid` transaction does not *return* the current winner, value of the highest bid, or number of bids to the caller, and the only reason it needs to *read* those values is to perform commutative MAX and ADD operations. Figure 7 shows the Doppel version of the transaction that exploits these observations. The new version uses the maximum bid in OPUT to choose the correct core's `maxBidder` value (the logic here says the highest bid should determine the value of that key). This changes the semantics of `StoreBid` slightly. In the original `StoreBid` if two concurrent transactions bid the same highest value for an auction, the first to commit is the one that wins. In Figure 7, if two concurrent transactions bid the same highest value for an auction at the same coarse-grained timestamp, the one with the highest core ID will win. Doppel can execute Figure 7 in the split phase.

Using the *top-K set* record type, Doppel can support inserts to contended lists. The original RUBiS benchmark does not specify indexes, but we use top-K sets to make browsing queries faster. We modify `StoreItem` to insert new items into top-K set indexes on `category` and `region`, and we modify `StoreBid` to insert new bids on an item into a top-K set index per item, `bidsPerItemIndex`. `SearchItemsByCategory`, `SearchItemsByRegion`, and `ViewBidHistory` read from these records. Finally, we modify `StoreComment` to use ADD on the `userRating`.

These examples show how Doppel's commutative operations allow seemingly conflicting transactions to be re-cast in a way that allows concurrent execution. This

pattern apppears in many other Web applications. For example, Reddit [2] also materializes vote counts, comment counts, and links per subreddit [3]. Twitter [1] materializes follower/following counts and ordered lists of tweets for users' timelines.

8 Evaluation

This section presents measurements of Doppel's performance, supporting the following hypotheses:

- Doppel increases throughput for transactions with conflicting writes to split data (§8.2).
- Doppel can cope with changes in which records are contended (§8.3).
- Doppel makes good decisions about which records to split when key popularity follows a smooth distribution (§8.4).
- Doppel can help workloads with a mix of read and write transactions on split data (§8.5).
- Doppel transactions which read split data have high latency (§8.6).
- Doppel increases throughput for a realistic application (§8.8).

8.1 Setup

All experiments are executed on an 80-core Intel machine with 8 2.4Ghz 10-core Intel chips and 256 GB of RAM, running 64-bit Linux 3.12.9. In the scalability experiments, after the first socket, we add cores an entire socket at a time. We run most fixed-core experiments on 20 cores.

The worker thread on each core both generates transactions as if it were a client, and executes those transactions. If a transaction aborts, the thread saves the transaction to try at a later time, chosen with exponential backoff, and generates a new transaction. backoff, and generates a new transaction. Throughput is measured as the total number of transactions completed divided by total running time; at some point we stop generating new transactions and then measure total running time as the latest time that any existing transaction completes (ignoring saved transactions). Each point is the mean of three consecutive 20-second runs, with error bars showing the min and max.

The Doppel coordinator changes the phase every 20 milliseconds. Doppel uses the technique described in §5.5 to determine which data to split. The benchmarks omit many costs associated with a real database; for example we pre-allocate all the records and do not incur any costs related to network, RPC, or disk.

In most experiments we measure phase reconciliation (Doppel), optimistic concurrency control (OCC), and two-phase locking (2PL). Doppel and OCC transactions abort and later retry when they see a locked item; 2PL uses Go's read-write mutexes. Both OCC and 2PL are implemented in the same framework as Doppel.

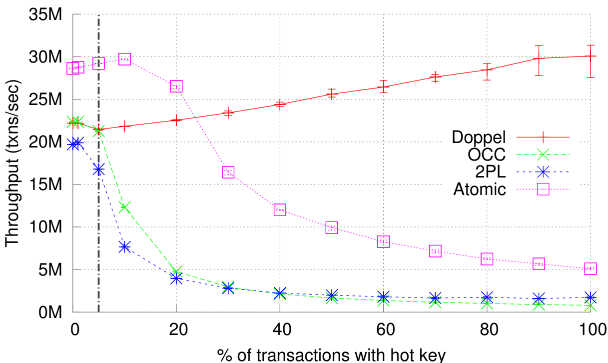


Figure 8: Total throughput for INCR1 as a function of the percentage of transactions that increment the single hot key. 20 cores. The vertical line indicates when Doppel starts splitting the hot key.

8.2 Parallelism versus Conflicts

This section shows that Doppel improves performance on a workload with many conflicting writes, using the following microbenchmark:

INCR1 microbenchmark. There are 1M 16-byte keys, and each transaction increments the value of a single key. There is a single popular key and we vary the percentage of transactions which increment that key; each other transaction randomly chooses from the not-popular keys.

This experiment compares Doppel with OCC, 2PL, and a system called Atomic. Doppel without split keys and OCC read the value of a key, compute the new value, and try to lock the key and validate that it hasn't changed since it was first read. If the key is locked or its version has changed, both abort the transaction and save it to try again later. 2PL waits for a write lock on the key, reads it, and then writes the new value. 2PL never aborts. Atomic uses an atomic increment instruction with no other concurrency control. Atomic represents an upper bound for locking schemes.

Figure 8 shows the throughputs of these schemes with INCR1 as a function of the percentage of transactions that write the single hot key.

At the extreme left of Figure 8, when there is little conflict, Doppel does not split the hot key, causing it to behave and perform similarly to OCC. With few conflicts, all of the schemes benefit from the 20 cores available.

As one moves to the right in Figure 8, OCC, 2PL, and Atomic provide decreasing total throughput. The high-level reason is that they must execute operations on the hot key serially, on only one core at a time. Thus their throughputs ultimately drop by roughly a factor of 20, as they move from exploiting 20 cores to doing useful

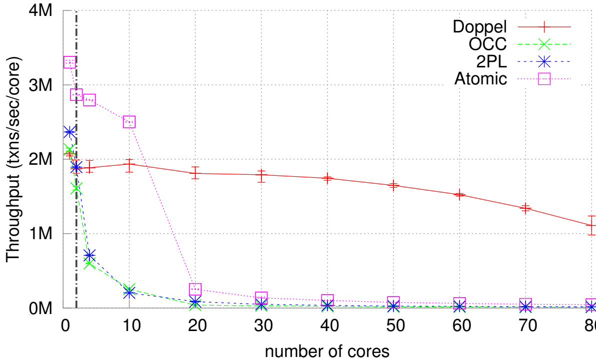


Figure 9: Throughput per core for INCR1 when all transactions increment a single hot key. The y-axis shows per-core throughput, so perfect scalability would result in a horizontal line.

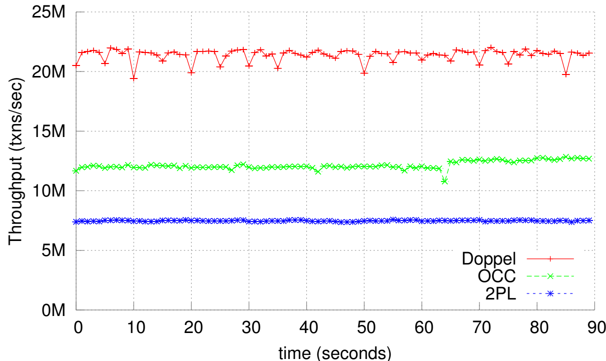


Figure 10: Throughput over time on INCR1 when 10% of transactions increment a hot key, and that hot key changes every 5 seconds.

work on only one core. The differences in throughput among the three schemes stem from differences in concurrency control efficiency: Atomic uses the hardware locking provided by the cache coherence and interlocked instruction machinery; 2PL uses Go mutexes which yield the CPU; while OCC saves and re-starts aborted transactions. The drop-off starts at an x value of about 5%; this is roughly the point at which the probability of more than one of the 20 cores using the hot item starts to be significant.

Doppel has the highest throughput for most of Figure 8 because once it splits the key, it continues to get parallel speedup from the 20 cores as more transactions use the hot key. Towards the left in Figure 8, Doppel obtains parallel speedup from operations on different keys; towards the right, from split operations on the one hot key. The vertical line indicates where Doppel starts splitting the hot key. Doppel throughput gradually increases as a smaller fraction of operations apply to non-popular keys, and thus a smaller fraction incur the DRAM latency required to fetch such keys from memory. When 100% of transactions increment the one hot key, Doppel performs 6.2× better than Atomic, 19× better than 2PL, and 38× better than OCC.

We also ran the INCR1 benchmark on Silo to compare Doppel's performance to an existing system. Silo has lower performance than our OCC implementation at all points in Figure 8, in part because it implements more features. When the transactions choose keys uniformly, Silo finishes 11.8M transactions per second on 20 cores. Its performance drops to 102K transactions per second when 100% of transactions write the hot key.

To illustrate the part of Doppel's advantage that is due to parallel speedup, Figure 9 shows multi-core scaling when all transactions increment the same key. The y-axis shows transactions/sec/core, so perfect scalability (perfect parallel speedup) would result in a horizontal line. Doppel falls short of perfect speedup, but nevertheless yields significant additional throughput for each core added. The lines for the other schemes are close to $1/x$ (additional cores add nothing to the total throughput), consistent with essentially serial execution. The Doppel line decreases because phase changes take longer with more cores; phase change must wait for all cores to finish their current transaction.

In summary, Figure 8 shows that when even a small fraction of transactions write the same key, Doppel can help performance. It does so by parallelizing update operations on the popular key.

8.3 Changing Workloads

Data popularity may change over time. Figure 10 shows the throughput over time for the INCR1 benchmark with 10% of transactions writing the hot key, with the identity of the one hot key changing every 5 seconds. Doppel throughput drops every time the popular key changes and a new key starts gathering conflicts. Once Doppel has measured enough conflict on the new popular key, it marks it as split. The adverse effect on Doppel's throughput is small since it adjusts quickly to each change.

8.4 Deciding What to Split

Doppel must decide whether to split each key. At the extremes, the decision is easy: splitting a key that causes few aborts is not worth the overhead, while splitting a key that causes many aborts may greatly increase parallelism. Section 8.2 explored this spectrum for a single popular key. This section explores a harder set of situations, ones in which there is a smooth falloff in the distribution of key popularity. That is, there is no clear distinction between hot keys and non-hot keys. The main question is whether Doppel chooses the right number (if any) of most-popular keys to split.

This experiment uses a Zipfian distribution of popularity, in which the kth most popular item is accessed in

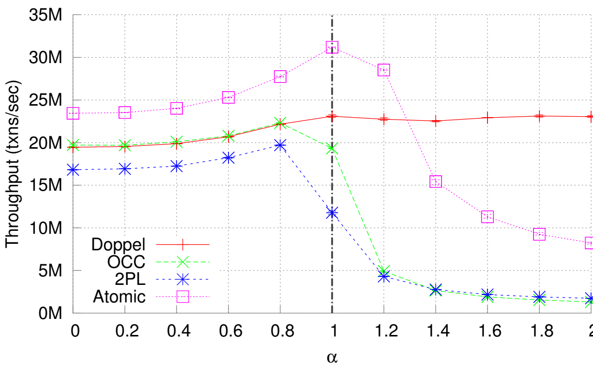


Figure 11: Total throughput for INCRZ as a function of α (the Zipfian distribution parameter). The skewness of the popularity distribution increases to the right. 20 cores. The vertical line indicates when Doppel starts splitting keys.

α	1st	2nd	10th	100th
0.0	.0001%	.0001%	.0001%	.0001%
0.2	.0013%	.0011%	.0008%	.0005%
0.4	.0151%	.0114%	.0060%	.0024%
0.6	.1597%	.1054%	.0401%	.0101%
0.8	1.337%	.7678%	.2119%	.0336%
1.0	6.953%	3.476%	.6951%	.0695%
1.2	18.95%	8.250%	1.196%	.0755%
1.4	32.30%	12.24%	1.286%	.0512%
1.6	43.76%	14.43%	1.099%	.0276%
1.8	53.13%	15.26%	.8420%	.0133%
2.0	60.80%	15.20%	.6079%	.0061%

Table 1: The percentage of writes to the first, second, 10th, and 100th most popular keys in Zipfian distributions for different values of α, 1M keys.

α	# Moved	% Reqs
< 1	0	0.0
1.0	2	10.5
1.2	4	35.9
1.4	4	56.1
1.6	4	70.5
1.8	4	80.1
2.0	3	82.7

Table 2: The number of keys Doppel moves for different values of α in the INCRZ benchmark.

proportion to $1/k^{\alpha}$. We vary α to explore different skews in the popularity distribution, using INCRZ:

INCRZ microbenchmark. There are 1M 16-byte keys. Each transaction increments the value of one key, chosen with a Zipfian distribution of popularity.

Figure 11 shows total throughput as a function of α. At the far left of the graph, key access is uniform. Atomic performs better than Doppel and OCC, and both better than 2PL, for the same reasons that govern the left-hand extreme of Figure 8.

As the skew in key popularity grows—for α values up to about 0.8—all schemes provide increasing throughput. The reason is that they all enjoy better cache locality as a set of popular keys emerge. Doppel does not split any keys in this region, and hence provides throughput similar to that of OCC.

Figure 11 shows that Doppel starts to display an advantage once α is greater than 0.8, because it starts splitting. These larger α values cause a significant fraction of transactions to involve the most popular few keys; Table 1 shows some example popularities. Table 2 shows how many keys Doppel splits for each α. As α increases to 2.0, Doppel splits the 2nd, 3rd, and 4th-most popular keys as well, since a significant fraction of the transactions modify them. Though the graph doesn't show this region, with even larger α values Doppel would return to splitting just one key.

In summary, for this workload Doppel does a good job of identifying which and how many keys are worth splitting, despite the gradual transition from popular to unpopular keys.

8.5 Mixed Workloads

This section shows how Doppel behaves when workloads both read and write popular keys. The best situation for Doppel is when there are lots of update operations to the contended key, and no other operations. If there are other operations on a split key, such as reads, Doppel's phases essentially batch writes into the split phases, and reads into the joined phases; this segregation and batching increases parallelism, but incurs the expense of stashing the read transactions during the split phase. In addition, the presence of the non-update operations makes it less clear to Doppel's algorithms whether it is a good idea to split the hot key. To evaluate Doppel's performance on a more challenging, but still understandable, workload, we use the LIKE benchmark from §7 that simulates users "liking" pages on a social networking site.

LIKE. The database contains a row for each user and a row for each page. Each transaction involves a user and a page. The user is always chosen uniformly at random. A write transaction chooses a page from a Zipfian distribution, increments the page's count of likes, and updates the user's row; the user's row is rarely contended, but the page's count might be. A read transaction chooses a page using the same Zipfian distribution, and reads the page's count and the user's row. There are 1M users and 1M pages, and unless specified otherwise the transaction mix is 50% reads and 50% writes.

Figure 12 shows throughput for Doppel, OCC, and 2PL with LIKE on 20 cores as a function of the fraction of transactions that write, with $\alpha = 1.4$. This setup causes the most popular page key to be used in 32% of transactions.

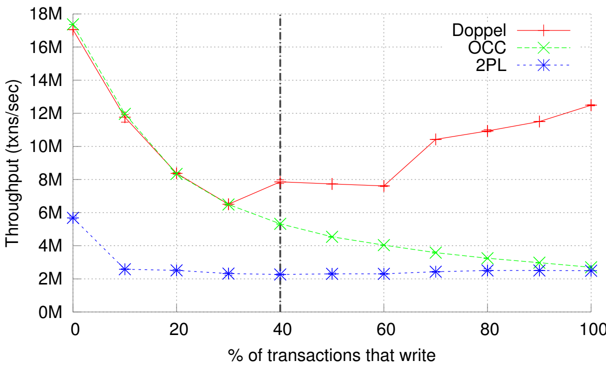


Figure 12: Throughput of the LIKE benchmark with 20 cores as a function of the fraction of transactions that write, $\alpha = 1.4$.

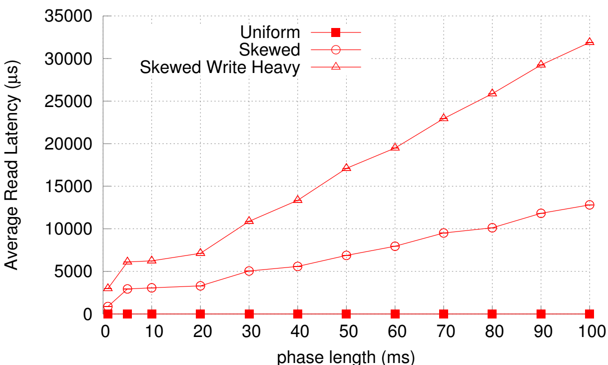


Figure 13: Average read transaction latencies in Doppel with the LIKE benchmark, varying phase length. A uniform workload, a skewed workload with 50% reads and 50% writes, and a skewed workload with 10% reads and 90% writes. 20 cores.

We would expect OCC to perform the best on a read-mostly workload, which it does. Until 30% writes Doppel does not split, and as a result performs about the same as OCC.

Doppel starts splitting data when there are 30% write transactions. This situation is tricky for Doppel because the split keys are read even more than they are written, so many read transactions have to be stashed. Figure 12 shows that Doppel nevertheless gets the highest throughput for all subsequent write percentages.

This example shows that Doppel's batching of transactions into phases allows it to extract parallel performance from contended writes even when there are many reads to the contended data.

8.6 Latency

Doppel stashes transactions which read split data in the split phase. This increases latency, because such transactions have to wait up to 20 milliseconds for the next joined phase. We use the LIKE benchmark to explore latency on two workloads (uniform popularity and skewed popularity with Zipf parameter $\alpha = 1.4$), separating latencies for read-only transactions and transactions that write. To measure latency, we measure the difference between the time each transaction is first submitted and when it commits. The workload is half read and half write transactions.

Table 3 shows the results. Doppel and OCC perform similarly with the uniform workload because Doppel does not split any data. In the skewed workload Doppel's write latency is the lowest because it splits the four most popular page records, so that write transactions that update those records do not need to wait for serial access to the data. Doppel's read latencies are high because reads of hot data during split mode have to wait up to 20 milliseconds for the next joined phase. This delay is the price Doppel pays for achieving almost twice the throughput of OCC.

8.7 Phase Length

When a transaction tries to read split data during a split phase, its expected latency is determined by the phase length; a shorter phase length results in less latency, but potentially lowered throughput. Figures 13 and 14 show how phase length affects read latency and throughput on three LIKE workloads. "Uniform" uses uniform key popularity and has 50% read transactions; nothing is split. "Skewed" has Zipfian popularity with $\alpha = 1.4$ and 50% read transactions; once the phase length is > 2ms, which is long enough to accumulate conflicts, Doppel moves either 4 or 5 keys to split data. "Skewed Write Heavy" has Zipfian popularity with $\alpha = 1.4$ and 10% read transactions; Doppel moves 20 keys to split data.

Figure 13 shows that the phase length directly determines the latency of transactions that read hot data and have to be stashed. Shorter phases are better for latency, but too short reduces throughput. The throughputs are low to the extreme left in Figure 14 because phase change takes about half a millisecond (waiting for all cores to finish split phase), so phase change overhead dominates throughput at very short phase lengths. For these workloads, the measurements suggest that the smallest phase length consistent with good throughput is five milliseconds.

8.8 RUBiS

Do Doppel's techniques help in a complete application? We measure RUBiS [6], an auction Web site implementation, to answer this question.

Section 7 describes our RUBiS port to Doppel. We modify six transactions to use Doppel operations; `StoreBid`, `StoreComment`, and `StoreItem` to use Max, Add, OPut, and TopKInsert, and `SearchItemsByCategory`, `SearchItemsByRegion`, and `ViewBidHistory` to read from *top-K set* records as indexes. This means Doppel can potentially mark

	Uniform workload			Skewed workload		
	Mean latency	99% latency	Txn/s	Mean latency	99% latency	Txn/s
Doppel	1µs R / 1µs W	1µs R / 2µs W	11.8M	1262µs R / 4µs W	20804µs R / 2µs W	10.3M
OCC	1µs R / 1µs W	1µs R / 2µs W	11.9M	26µs R / 1069µs W	22µs R / 1229µs W	5.6M
2PL	1µs R / 1µs W	2µs R / 2µs W	9.5M	1µs R / 8µs W	3µs R / 215µs W	3.7M

Table 3: Average and 99% read and write latencies for Doppel, OCC, and 2PL on two LIKE workloads: a uniform workload and a skewed workload with $\alpha = 1.4$. Times are in microseconds. OCC never finishes 156 read transactions and 8871 write transactions in the skewed workload. 20 cores.

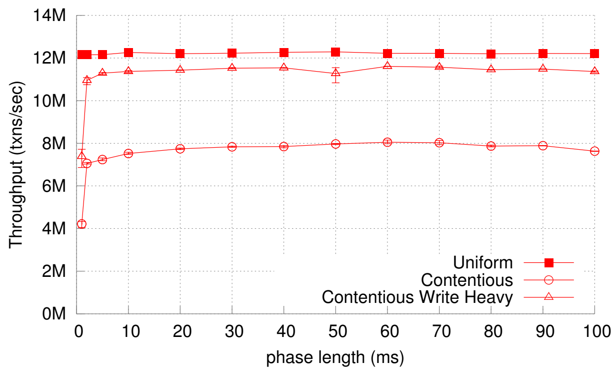


Figure 14: Throughput in Doppel with the LIKE benchmark, varying phase length. A uniform workload, a skewed workload with 50% reads and 50% writes, and a skewed workload with 10% reads and 90% writes. 20 cores.

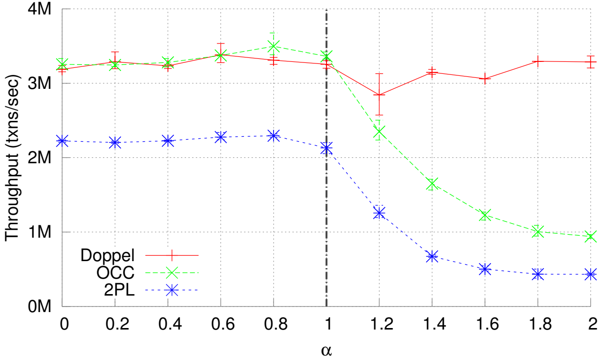


Figure 15: The RUBiS-C benchmark, varying α on the x-axis. The skewness of the popularity distribution increases to the right. 20 cores.

	RUBiS-B	RUBiS-C
Doppel	3.4	3.3
OCC	3.5	1.1
2PL	2.2	0.5

Table 4: The throughput of Doppel, OCC, and 2PL on RUBiS-B and on RUBiS-C with Zipfian parameter $\alpha = 1.8$, in millions of transactions per second. 20 cores.

auction metadata as split data. The implementation includes only the database transactions; there are no web servers or browsers.

We measured the throughput of two RUBiS workloads. One is the Bidding workload specified in the RUBiS benchmark, which consists of 15% read-write transactions and 85% read-only transactions; this ends up producing 7% total writes and 93% total reads. We call this RUBiS-B. In RUBiS-B most users are browsing listings and viewing items without placing a bid. There are 1M users bidding on 33K auctions, and access is uniform, so when bidding, most users are doing so on different auctions. This workload has few conflicts and is read-heavy.

We also created a higher-contention workload called RUBiS-C. 50% of its transactions are bids on items chosen with a Zipfian distribution and varying α. This approximates very popular auctions nearing their close. The workload executes non-bid transactions in correspondingly reduced proportions.

Table 4 shows how Doppel's throughput compares to OCC and 2PL. The RUBiS-C column uses a somewhat arbitrary $\alpha = 1.8$. As expected, Doppel provides no advantage on uniform workloads, but is significantly faster than OCC and 2PL when updates are applied with skewed record popularity.

Figure 15 explores the relationship between RUBiS-C record popularity skew and Doppel's ability to beat OCC and 2PL. Doppel gets close to the same throughput up to $\alpha = 1$. Afterwards, Doppel gets higher performance than OCC. When $\alpha = 1.8$ Doppel gets approximately 3× the performance of OCC and 6× the performance of 2PL.

Doppel's techniques make the most difference for the `StoreBid` transaction, shown in Figures 6 and 7. Doppel marks the number of bids, max bid, max bidder, and the list of bids per item of popular products as split data. It's important that the programmer wrote the transaction in a way that Doppel can split all of these data items; if the update for any one of the items had been programmed in a non-splittable way (e.g., with explicit read and write operations) Doppel would execute the transactions serially and get far less parallel speedup.

In Figure 15 with $\alpha = 1.8$, OCC spends roughly 67% of its time running `StoreBid`; much of this time is consumed by retrying aborted transactions. Doppel eliminates almost all of this 67% by running the transactions in parallel, which is why Doppel gets three times as much throughput as OCC with $\alpha = 1.8$.

These RUBiS measurements show that Doppel is able to parallelize substantial transactions with updates to multiple records and, skew permitting, significantly outperform OCC.

9 Conclusion

Doppel is an in-memory transactional database which uses phase reconciliation to increase throughput. The key idea is to execute certain types of conflicting operations on local per-core data, in parallel, and to reconcile the per-core states periodically. On workloads with many writes to a small number of popular records, Doppel can increase throughput by a factor related to the number of available cores.

Acknowledgments

We thank the anonymous reviewers, and our shepherd Allen Clement, for their helpful feedback. This research was supported by NSF awards 1065114, 1302359, 1301934, 0964106, 0915164, by Quanta, and by Google.

References

[1] Cassandra @ Twitter: An interview with Ryan King. `http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king`.

[2] Reddit. `http://reddit.com`.

[3] Reddit codebase. `https://github.com/reddit/reddit`.

[4] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record*, 24(2):23–34, 1995.

[5] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[6] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *International Workshop on Workload Characterization*, pages 3–13. IEEE, 2002.

[7] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *High Performance Embedded Architectures and Compilers*, pages 4–18. Springer, 2009.

[8] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[9] P. A. Bernstein, C. W. Reid, and S. Das. Hyder—a transactional record manager for shared flash. In *CIDR*, volume 11, pages 9–20, 2011.

[10] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *PVLDB*, 4(11):944–955, 2011.

[11] S. Boyd-Wickizer. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, 2013.

[12] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Eurosys*, pages 211–224. ACM, 2013.

[13] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP*, pages 1–17. ACM, 2013.

[14] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254. ACM, 2013.

[15] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.

[16] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *PVLDB*, 23(1):1–23, 2014.

[17] A. Kemper and T. Neumann. HyPer: A hybrid OLTP and OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206. IEEE, 2011.

[18] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[19] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[20] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, 2014.

[21] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing*, 2007.

[22] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278. USENIX Association, 2012.

[23] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2):928–939, 2010.

[24] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *Eurosys*, pages 17–30. ACM, 2011.

[25] J. Schneider, F. Landau, and R. Wattenhofer. Synchronization phases (to speed up transactional memory). Technical report, July 2011.

[26] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[27] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400. ACM, 2011.

[28] M. Stonebraker, S. Madden, J. D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[29] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Engineering Bulletin*, 36(2), 2013.

[30] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32. ACM, 2013.

[31] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

Eidetic Systems

David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen
University of Michigan

Abstract

The vast majority of state produced by a typical computer is generated, consumed, then lost forever. We argue that a computer system should instead provide the ability to recall any past state that existed on the computer, and further, that it should be able to provide the lineage of any byte in a current or past state. We call a system with this ability an *eidetic computer system*. To preserve all prior state efficiently, we observe and leverage the synergy between deterministic replay and information flow. By dividing the system into groups of replaying processes and tracking dependencies among groups, we enable the analysis of information flow among groups, make it possible for one group to regenerate the data needed by another, and permit the replay of subsets of processes rather than of the entire system. We use model-based compression and deduplicated file recording to reduce the space overhead of deterministic replay. We also develop a variety of linkage functions to analyze the lineage of state, and we apply these functions via retrospective binary analysis. In this paper we present Arnold, the first practical eidetic computing platform. Preliminary data from several weeks of continuous use on our workstations shows that Arnold's storage requirements for 4 or more years of usage can be satisfied by adding a 4 TB hard drive to the system.[1] Further, the performance overhead on almost all workloads we measured was under 8%. We show that Arnold can reconstruct prior state and answer lineage queries, including backward queries (on what did this item depend?) and forward queries (what other state did this item affect?).

1 Introduction

The vast majority of state produced by a typical computer is generated, consumed, then lost forever. Lost state includes process address spaces, deleted files, interprocess communication, and input received from the network. With lost state comes lost value: users cannot recover detailed information about past computations that would be useful for auditing, forensics, debugging, error tracking, and many other purposes.

Prior approaches try to retain some of this information via a variety of techniques, such as file backup, packet logging, and process checkpointing, but these approaches preserve only the subset of information that someone anticipates may be useful. A more comprehensive approach is needed: one that preserves the values and lineage of *all* state that has ever existed on the system. We call such a system an *eidetic computer system*.

An eidetic computer system can recall any past state that existed on that computer, including all versions of all files, the memory and register state of processes, interprocess communication, and network input. Further, an eidetic computer system can explain the lineage of each byte of current and past state.

Lineage describes *how* state was derived. With such information, the user of an eidetic system can often infer *why* the data was derived. For instance, a colleague might point out to a user that a citation in a paper draft is incorrect. Using an eidetic system, the user could trace back from the binary document through all the steps used to create that document and recreate the browser screen displaying the Web page from which the data was derived. On seeing that Web page, the user would realize that he cited the wrong paper from a conference session. The user could then trace forward from that mistake and reveal all current documents and data that reflect the mistake, as well as any external output (e.g., e-mail) containing mistaken information.

Or consider an example in which someone runs a malicious application on a shared computer. The malicious program exploits a privilege escalation vulnerability, gives itself privileged access, and installs a backdoor for future access. An eidetic system could trace forward from the malicious program, trace through the priv-

[1] Currently, a 4 TB drive can be purchased for approximately $150.

ilege escalation vulnerability, and determine that the malicious software installed a backdoor. The system could then trace any future executions of the vulnerable program and determine if the backdoor was ever used, and exactly what was done by the attacker during the vulnerable window. In these and similar examples, recall and lineage are tightly coupled; they are useful in isolation but much more powerful when combined.

In this paper, we describe an eidetic system called Arnold that provides the above properties for personal computers and workstations with reasonable storage requirements and runtime overheads. The key technologies that enable Arnold to provide the properties of an eidetic system efficiently are deterministic record and replay [10], model-based compression, deduplicated file recording, operating system tracking of information flow between processes [23], and retrospective binary analysis of information flow within processes [11, 35].

Arnold uses deterministic record and replay to efficiently reproduce past computations. Reproducing past computations enables Arnold to recall any state and to track the lineage of that state within a replaying entity. Arnold uses numerous optimizations to reduce the amount of data that must be recorded. As a result, the log data required for years of operation of a personal computer or workstation can fit on a commodity hard drive.

To avoid the need to replay the entire system to recover any state, Arnold divides the system into units, called *replay groups*, that can be replayed independently. To track information flow between replay groups, Arnold records dependency information for each communication between replay groups, forming a *dependency graph*. In addition to enabling information flow to be tracked across groups, the dependency graph also allows Arnold to treat as a cache the log of data sent between groups. To conserve space, Arnold can discard this data and regenerate it later by replaying the group that produced it, a technique we call *cooperative replay*.

To analyze lineage within a replay group, Arnold uses retrospective binary analysis, in which it deterministically reexecutes the processes within the group and tracks the relationships between inputs and outputs. Different *linkage functions* may be used to define dependencies according to how the lineage analysis will be used. Arnold defers the choice of linkage function to the time of the query. This preserves flexibility and enables it to answer lineage queries that were not anticipated during the original execution. It also moves the overhead of analysis from original execution to the time of query.

Putting these pieces together, Arnold can answer both backward queries (where did this particular state come from?) and forward queries (what outputs and current state are derived from this prior state?). It does so by following the dependency graph, reexecuting the process groups to learn how their inputs map to their outputs, and querying the graph to learn how group outputs map to the inputs of other groups.

Underlying Arnold's design is the observation that deterministic replay and information flow are synergistic. Recording information flow among processes saves storage space by eliminating the need to record the data sent between processes. Deterministic replay makes it possible to reproduce any transient state of a prior process execution. This makes it possible to perform information flow queries over that state that were not imagined at the time the process executed. It also provides the ability to defer the work of tracking information flow within processes until the results are needed.

We have run Arnold continuously on our workstations for several weeks. Our results show that its storage requirements for 4 or more years of operation could be satisfied by adding a $150 4 TB hard drive. On almost all benchmarks we ran, Arnold's performance overhead is less than 8%. We also report on several case studies in which we use Arnold to reproduce past state and trace lineage over many applications and workflows.

2 Related work

Arnold draws upon prior research from many areas. Many systems have sought to save some prior state in a system. For example, versioning file systems [39, 44, 49] store regular snapshots of file state; process checkpointing systems [40] store snapshots of running processes; and systems like DejaView snapshot both processes and files [24]. These systems store only a subset of the state in a system, and they take checkpoints only at coarse-grained points in time to reduce storage usage. Checkpoints are insufficient to reproduce computation or to track intra-process lineage. In contrast, Arnold can reproduce all state and computation in a system at the granularity of individual instructions.

Arnold uses deterministic record and replay to reproduce all state and computation in a system. Deterministic replay for uniprocessors is a mature technology, and implementations exist in both software [54, 10, 14, 17, 36, 43, 46] and hardware [6, 34, 53, 21, 30, 33, 51]. Making deterministic replay efficient on multi-processors is an ongoing research challenge [15, 25, 50, 3, 38, 52, 55, 12, 26], as is making execution on multiprocessors deterministic [8, 9, 13, 28, 37]. We deterministically replay a multiprocessor system by recording synchronization operations and instrumenting races, but this is not a focus of this research. Instead, we focus on applying deterministic record and replay to build an eidetic system and on reducing the storage overhead for long-term use through techniques such as model-based compression.

Checkpointing and rollback-recovery have often been

used to restore past state [16]. However, the focus of most prior work has been to tolerate failures by reproducing the latest correct state, rather than to restore any past state as in eidetic systems.

Prior research has examined how to track the lineage of data, either within a process [35] or between processes [23, 22, 18]. Provenance-aware storage systems [31, 32] annotate file data with causal history to capture the relationship between processes and files. Arnold tracks lineage within a process, between processes, and between files and processes. Unlike prior systems, Arnold tracks comprehensive lineage for arbitrary executables and non-deterministic programs.

Other projects have examined how to index and query prior system state. DejaView [24] indexes and provides a query interface for prior information that is displayed on the screen or available through the accessibility API. Tralfamadore traces the execution of a system and provides mechanisms and components to analyze that trace [27]. Arnold provides the ability to reproduce all state and track its lineage across arbitrary computations.

Our technique to regenerate inter-group communication via replay trades storage for recomputation. Other projects have made a similar tradeoff in different domains [19, 2, 7, 47]. For example, Nectar [19] trades storage for computation in data-parallel cloud environments and supports recomputation of storage results from inputs and memoization of partial and full computations. While Nectar is restricted to DryadLINQ applications, which are both deterministic and functional, Arnold provides these and other benefits for general-purpose computation.

3 Design goals

The design of Arnold was guided by several goals. First, we wanted to support the widest possible range of queries about user-level state and the lineage of that state. Arnold reproduces and tracks the lineage of state of all user-level processes at the level of the instruction set architecture. We wanted to support queries (Section 4.8) about backward lineage (what influenced this data?) and forward lineage (what did this data influence?) both within a replay group (Section 4.6) and between replay groups (Section 4.5). We also wanted to support queries not anticipated at the time of recording, which we accomplish via retrospective binary analysis (Section 4.6).

Second, we wanted to minimize the time and space overhead of recording, since we intend for Arnold to continuously record computer usage. We wanted the time overhead of recording to be low enough to support interactive workloads and the space overhead to be small enough to record several years of execution of workstations and personal computers on a commodity hard drive. We reduce the time overhead of recording through deterministic record and replay (Section 4.1) and retrospective binary analysis (Section 4.6). We reduce space overhead through techniques such as model-based compression (Section 4.2), deduplicated file recording (Section 4.3), and cooperative replay (Section 4.4).

Third, we wanted to reduce the cost of answering queries by not requiring the reexecution of processes unrelated to the state being queried. We accomplish this by dividing the system into multiple replay groups, each of which can be replayed independently. To preserve lineage between replay groups, we track the dependencies cause by inter-group communication in a dependency graph (Section 4.5).

4 Design and implementation

4.1 Record and replay

Deterministic record and replay enables two important features of Arnold. First, it allows Arnold to efficiently reproduce the complete architectural state (register and address space) of user-level processes. Second, it allows Arnold to defer the work needed to track lineage from the time of execution to the time of querying [11].

To enable reproduction of all architectural state, Arnold records and replays execution at the level of processes. Our modified Linux kernel records all nondeterministic data that enters a process: the order, return values, and memory addresses modified by a system call; the timing and values of received signals; and the results of querying the system time.

Dealing with multiple threads/processes that write-share memory requires special care. Record and replaying individual threads/processes would shrink the scope of replay needed to answer a query, but this would require Arnold to record all nondeterministic reads of shared memory. Instead, Arnold records all threads/processes that share memory as a single *replay group*, then seeks to replay the interleavings of events from the replay group deterministically.

To enable deterministic replay of a replay group, Arnold records all synchronization operations and atomic hardware instructions (such as `atomic_inc`, or `atomic_dec_and_test`). A modified version of `libc` logs the order and memory addresses of synchronization operations between threads, including low-level atomic instructions and high-level synchronization operations such as `pthread_lock`. Such logging inserts an additional two atomic instructions for each event logged (to order the start and end of the operation). In the absence of data races, this information is sufficient to faithfully replay the recorded execution of a replay group involv-

ing multiple threads or processes—each replayed thread will execute the same sequence of instructions and system calls, observe the same values read, and produce the same results as during recording [42].

In the presence of data races, the replayed execution may diverge from the recorded one. We deal with programs with data races by identifying the races and adding additional instrumentation to eliminate them on subsequent runs. Veeraraghavan et al. [48] observed a synergy between deterministic replay and data race detection: if the only reason that a replayed execution may diverge from a recorded execution is the presence of a data race, then the replay system can act as a very efficient data-race detector. Arnold supports the ability to instrument and observe the execution of replayed recordings (Section 4.6), and we use this to run a standard vector-clock data race detector [41] when a replay divergence is detected. This is guaranteed to detect at least the first pair of racing instructions (it may also detect subsequent pairs). We then either statically instrument the code to record the outcome of the data race, or dynamically instrument the binary when it runs to cause the racing pair of instructions to trap to the kernel (via an `INT 3` instruction), where we record the order of the racing instructions. Static instrumentation is preferred since it is more efficient, but dynamic instrumentation allows us to support applications for which we do not have source code.

In practice, we have detected few data races that affect replay in the programs we run on our workstations. It has been relatively simple for a small team of users to add the necessary instrumentation to record these instances. Interestingly, many of the races we found were already documented, for example by developers who ran ThreadSanitizer [45] or similar tools. Since races are very infrequent, we suspect that it should usually be possible to search through all possible interleavings of the racing instructions to find an interleaving that is indistinguishable from the recorded execution [3, 38].

When a process executes the `exec` system call, Arnold creates a new replay group (with a unique 64-bit identifier) consisting solely of that process. Arnold also saves a small checkpoint for the new group, which allows replay to begin from the creation of that process. The checkpoint consists of the arguments and environment variables passed to `exec`, other nondeterministic information used during the system call (e.g., seeds used to randomize address spaces), and a *reference* to the file containing the executable image—the image usually resides in a deduplicated file store described in Section 4.3.

Arnold creates new replay groups on `exec` rather than on `fork` because the initial address space at `exec` is more amenable to deduplication than the address space at the time of `fork`. It stores a *split record* that contains the unique identifier of the new replay group in the log of the replay group that performed the `exec`. Infrequently, two replay groups need to be merged (e.g., because they establish a write-shared memory segment). In such instances, Arnold merges the processes from one group into the other and inserts a *merge record* into their logs.

Arnold replays recorded execution on a per-group basis. It creates a new process from the group's checkpoint and deterministically reexecutes the process by supplying values from the group's log in lieu of performing any nondeterministic action. As additional threads and processes are created within the replay group, Arnold also replays those entities. Each process executes until it exits or the execution reaches a split record. Arnold can replay multiple groups concurrently—this allows it to parallelize lineage queries that span groups.

4.2 Reducing storage utilization

Arnold uses several optimizations to reduce the size of its replay logs. The first optimization is model-based compression. The order and results of many of the system calls and synchronization operations that Arnold logs are highly predictable. For instance, many system calls usually return zero (success); the `write` system call usually returns the number of bytes in the input buffer; and `pthread_cond_lock` usually returns a value specifying that the lock has been obtained. Arnold constructs a model for predictable operations and records only instances in which the returned data differs from the model. Thus, the log size used for each type of operation is proportional to the number of deviations, which can be much less than the number of executed operations.

Some operations such as `poll` exhibit considerable locality in the data they return (e.g., the set of ready file descriptors is often the same from call to call within a short window). For these operations, Arnold caches the most recent 8 values returned on both record and replay and replaces the actual values in the log with a small cache index (when the value hits in the cache) in order to save space. Arnold also uses model-based compression to reduce the amount of ordering information in the log. It predicts that there are no ordering constraints and no signals delivered between two successive logged operations, and records only when the execution deviates from the model.

After applying model-based compression, we determined that the most significant source of log usage on our systems was messages sent from the X server to applications. A small fraction of this data comes from user events (button presses, mouse movements, etc.). Most data consisted of responses to application requests. Since such responses included nondeterministic data such as identifiers and window properties, the responses needed to be recorded to faithfully replay each application.

We observed, however, that with the exception of ac-

tual user input, the behavior of the X server is mostly deterministic. Arnold avoids logging most data from the X server by using the X server to help regenerate data during replay. We insert a proxy between applications and the X server that records only a small subset of the data sent from the X server, such as identifiers and window properties generated nondeterministically by the X server. During replay, the application again connects to an X server via the proxy. The proxy translates the nondeterministic values, and the replay process generates GUI state using the live X server, but on a separate display. The proxy also inserts the recorded user events at the appropriate point in the stream. In combination with the proxy translation, the X server produces the same sequence of responses during the replayed execution as during recording. With deterministic X recording, Arnold can make the display of X windows visible during replay. As we will describe, this is useful for showing users application displays that correspond to the results of lineage queries and for allowing users to specify queries by clicking on recreations of windows displaying data they observed in the past. By recording only nondeterministic response values and user input, the proxy substantially reduces the amount of information in the logs of GUI applications.

After applying the above optimizations, we noticed that time queries constituted a substantial portion of the remaining log size. To reduce the amount of nondeterminism that needs to be logged, Arnold uses a *semi-deterministic clock*. The value returned by a semi-deterministic clock is guaranteed to be less than the real-time clock for the system, and within a specified delta. The default delta is 10 ms; it may be overridden by applications that need more accuracy. A replay group's semi-deterministic clock is incremented deterministically based on the number and type of logged operations (which is the same during both recording and replay). When the time is queried, Arnold reads the actual real-time clock. If the semi-deterministic clock is greater than or more than delta behind the real-time clock, Arnold returns the real-time clock value, sets the semi-deterministic clock equal to the real-time clock, and records the new value in the log. Thus, the amount of time query data in the log is proportional to the number of such resets rather than the total number of time queries; if Arnold usually predicts the clock value correctly, the amount of logged time data can be quite small.

Arnold ensures that observed semi-deterministic clock values are externally consistent. It is for this reason that the semi-deterministic clock must always be less than the real-time clock. If a recorded process sends a message to a non-recorded process, the receiver will always observe that the message arrived after it was sent. Further, if a recorded process receives data from or sends data to an entity outside the replay group, the group's semi-deterministic clock is set to match the real-time clock. Thus, the observed clock values are causally consistent both across all processes on the computer system and with respect to external entities.

Finally, Arnold compresses all log data with `gzip`. This is very effective in compressing some input, such as text. It also helps to compress applications that perform repetitive operations with similar results.

4.3 Copy-on-RAW file cache

Arnold records the file data read by a process so that data can be redelivered to the process during replay. Recording this data can take a substantial amount of log space, so Arnold optimizes how the read file data is stored by deduplicating it. This works particularly well when a file is read multiple times before being modified.

To deduplicate the read file data, Arnold saves a version of a file only on the first read after the file is written. Subsequent reads log only a reference to the saved version, along with the read offset and return code. We refer to this as *copy-on-RAW (read-after-write) recording*.

If another process opens the file for writing while a reading process is running, the reading process reverts back to recording the read values instead of the reference to the stored version (several optimizations are possible here, such as recording the file version on each `read` instead of `open`, or reexecuting reads and writes to files shared among processes in the same replay group.)

Arnold also uses the copy-on-RAW store for file `mmap` operations by mapping the stored file version into the process space on replay. If the mapped region is writable, Arnold creates a private temporary copy of the file version on replay; this allows the replayed process to change the file contents without affecting other replayed processes that reference the same file version.

Note that, with this design, the current version of all files is stored in the default file system (ext4 on our Ubuntu workstations). We chose this operation for efficiency; recording processes (the common usage case) go through the well-optimized file system and receive the best performance. Copy-on-RAW population of the file store can proceed asynchronously and not slow down the recording process too much unless large amounts of data being read exert memory pressure. The cost of this implementation is some double-buffering of current file data, which we could reduce in the future.

We were initially surprised because the size of Arnold's file store grew more slowly than expected on our workstations. On investigation, we realized this was due to an important difference between Arnold's file store and a versioning file system: Arnold's file store does not have to store data that is written but never read. Since Arnold is an eidetic system, it can, of course, recre-

ate this file data; however, it does so by replaying the process(es) that produced the data rather than by retrieving the data from the file system. In contrast, a versioning file system needs to store all file versions even if they are overwritten or deleted without being read.

Since Arnold can reproduce any current or past file version via replay, it is the logs of nondeterminism that are Arnold's truly persistent store [16]. We can thus treat Arnold's copy-on-RAW file store as a *cache*. The copy-on-RAW file store (and, in fact, all file system data) is simply a performance optimization that contains checkpoints of data that could be produced by replay. This reasoning led us to develop *cooperative replay*.

4.4 Cooperative replay

We normally think of replay groups as independent entities: we log their nondeterministic inputs during recording and reinsert these inputs during replay. Cooperative replay provides another option, which is to use one replay group to regenerate the data read by another. Cooperative replay allows us to treat the log of all interprocess communication (files, pipes, etc.) as a cache, whose records can be evicted when the cache is full and recovered when needed during replay.

Arnold uses cooperative replay to regenerate data read from files. During replay, if the requested data exists in the file system (because it is the current version of the file) or in the copy-on-RAW file cache, Arnold reads the data from one of those locations. If not, Arnold regenerates the data by replaying the replay group(s) that produced the data. Arnold stores information about the source of all file data in each read record—this includes the identifier of the replay group(s) and the system call(s) executed by the group(s) that produced the file data (Section 4.5). To regenerate the data, Arnold suspends the replay group requesting the data, replays the producing replay group(s), repopulates any data evicted from the file cache, and finally resumes the requesting replay group.

Cooperative replay may recurse in a depth-first manner. When replaying replay group A, Arnold may need to replay another replay group B to regenerate file data read by replay group A, and this may trigger the replay of a third replay group C, and so forth. The recursion will stop when a group can be replayed without depending on any other replay group.

As data flows forward, it creates a directed acyclic graph. While no cycles exist between nodes in the graph, Arnold may encounter a scenario where two replay groups depend on outputs of each other. In this scenario Arnold will alternate replaying each group until all dependencies are met.

4.5 Dependency graph

To support cooperative replay and track lineage across replay groups, Arnold maintains a logical graph of the data-flow dependencies between groups, which we refer to as the *dependency graph*. Nodes in the graph are <replay group id, system call id> tuples, where the second part of the tuple uniquely identifies a particular system call executed by a process in the replay group. Each edge in the graph is a bidirectional link between the system call that produced data and the set of one or more system calls that consumed that data. Thus, Arnold can determine the lineage of data across replay groups by tracing backward in time through the dependency graph, and it can determine what downstream values were influenced by particular data by tracing the lineage forward.

We first describe the operation of the dependency graph for file data. When a recorded process writes to a file, Arnold records which bytes were modified, along with the <replay group id, system call id> in a per-file B-tree indexed by the file offset. The root of each per-file B-tree is in turn indexed in a B-tree of all files; we refer to this collection of B-trees as the *filemap*. Arnold allocates a separate region on disk for the filemap; it reads pages on demand into a kernel cache in physical memory and evicts pages using an LRU algorithm. Pages are flushed asynchronously using the journal mechanisms of the underlying file system (ext4 in our current implementation). Thus, the filemap contains the lineage information for all current file data in the file system.

When a recorded process reads from a file, Arnold searches through the filemap to find which system call(s) wrote the bytes being read. It copies the tuples out of the filemap into the replay log of the reading process. Thus, the log contains sufficient data to answer backward lineage queries (how was the data read by this process produced?). In order to answer forward lineage queries, Arnold generates an index over the reverse linkages and stores it in a `sqlite` database. A daemon process asynchronously generates the index by incrementally scanning recent replay logs (replay is unnecessary because the data needed to generate the index is in the logs).

Arnold uses a similar process to record the lineage of other forms of IPC. For pipes and sockets, it keeps metadata for bytes written but not yet consumed in the kernel. For most pipes and sockets, there is a single writing process and a single reading process, and bytes are read in the order they are written. In this common case, Arnold reduces log size by only logging the identifier of the writing replay group. On a query, Arnold identifies the system call(s) that generated data by scanning the log of the writing record group. If there is more than one reader or writer, Arnold tracks the reads and writes on the pipe or socket in the same manner as for file system data.

Arnold also tracks lineage of the data passed from the

parent process to the child process during exec. This includes arguments, environment variables, and some miscellaneous data used during the exec system call.

Arnold does not record the lineage of data passed among processes via shared memory. Instead, Arnold tracks this lineage at query time by instrumenting the memory read and write instructions as described in the next section.

4.6 Intra-process lineage

Arnold uses Pin [29] binary instrumentation to analyze replayed executions and track the lineage of data within a replay group. We chose Pin because it is a flexible and well-documented tool; however, Pin can be slow, partially because it dynamically, rather than statically, inserts instrumentation into running binaries. Arnold avoids overhead during recording by only using Pin and analyzing intra-process lineage during replay.

While analysis tools such as Pin are typically invisible to the program they instrument, they are not transparent to the operating system: such tools insert new system calls, allocate additional memory, catch signals, etc. Without special care, these extra actions to support analysis will cause the replayed execution to diverge from the recorded execution. Arnold uses techniques from X-ray [4] to compensate for the divergences caused by analysis; for instance, it prevents Pin from allocating memory that will conflict with the replayed execution and it identifies system calls generated by Pin and executes them live rather than trying to supply nondeterministic values from the group's log.

Arnold traces lineage within a group by restoring the group's checkpoint and replaying the processes within the group with Pin dynamic analysis enabled. Pin tools determine which inputs to the group influenced which outputs according to a customizable *linkage function*.

There are many possible ways of defining lineage: one can say that an input influences an output only if the output is derived from the input via a series of copies, or one can consider other forms of data flow, or control flow, etc. The linkage function defines, for each type of processor instruction, which outputs of the instruction are influenced by which inputs. Each linkage is implemented as a Pin tool. Arnold provides several common linkage functions (and applications may define their own):

- **Copy.** An input of an instruction influences an output only if the instruction copies the value of the input to the output location (e.g., via a move instruction).
- **Data flow.** An input of an instruction influences an output if the instruction uses the input to calculate the value of the output (e.g., via an add instruction).
- **Index.** An input influences an output if the input is used to calculate the output or if the input is used as an index to load a value used to calculate the output (e.g., via an array or lookup table index).
- **Control flow.** This includes, in addition to index and data flow influence, the influence propagated via control flow as tracked using the algorithms developed by ConfAid [5].

The lineage data returned by each linkage function above is a superset of the preceding linkage functions. For example, if a group input and output are related via the data flow linkage, they are also related via the index and control flow linkages.

A lineage query for a group specifies a set of inputs, a set of outputs, and a linkage function. Inputs may be specified as a set of <system call id, byte range> tuples, where each tuple denotes a unique system call performed by the recorded group and the subset of input bytes to track for that call. Alternatively, the input may be specified as a class of input (e.g., all file data or all GUI events from the X server), or the query may simply track all inputs. Output is specified similarly.

Arnold uses taint tracking to derive intra-group lineage. When it sees a system call matching the input specification, it taints the requested bytes with unique identifiers as they are read into the process address space. As instructions execute, the Pin lineage tool propagates that taint among memory addresses and registers according to the linkage function. When Arnold sees an output matching the specification, it writes the taint of each output byte to a results file. Of course, each byte may be influenced by zero to many inputs.

4.7 User-propagated lineage

For interactive workflows, lineage may pass through the user of the system. For instance, she might view a Web page in a browser, then type text from that page into an editor. Arnold tracks such lineage by first identifying inputs and outputs that occurred at approximately the same time, then using fuzzy string matching to look for contextual linkages among those inputs and outputs.

Arnold identifies user-generated inputs with a Pin tool that runs on a replayed execution. The tool identifies channels corresponding to user input: sockets used to communicate with the X server (for GUI input), the terminal device (for text input), and network sockets connecting to well-known ports associated with user input (e.g., the sshd port). It generates a temporary file containing the stream of data read from every such channel read by the replay group. The tool performs channel-specific parsing: for instance, it decodes the X messages to read the corresponding key press events, and it intercepts data returned from functions such as SSL_read to retrieve the unencrypted data from ssh sockets. The channel input stream file therefore contains textual data that corresponds to the input.

Arnold follows a similar strategy to generate output stream files for a replay group. Understanding GUI output turned out to be tricky, however, because most programs we looked at did not send text to the X server, but instead sent binary glyphs generated by translating the output characters into a particular font. Arnold identifies these glyphs as they are passed to standard X and graphical library functions. It traces the lineage backward from these glyphs using one of the above linkages (e.g., the index linkage). These values are typically influenced by either or both of (a) textual input to the process being reexecuted, or (b) standard font files. By tracing glyphs to either location, Arnold recovers the characters associated with those glyphs. If the lineage is traced to a font file Arnold must determine the font character from the font file. This requires Arnold to understand font file formats (of which there are relatively few). Thus, it translates the sequence of glyph outputs to the underlying text they depict.

4.8 Query Execution

Arnold queries allow users to recall prior state and lineage information. There are three types of queries:

- **State queries** retrieve past transient state, persistent state, inputs, or outputs of the computer system.
- **Backward lineage queries** start at any current or past state, input, or output and trace the lineage of the bytes comprising that state backward in time according to a specified linkage function. A backward query answers the question: *How was this state derived?*
- **Forward lineage queries** start at a past state, input, or output and trace the lineage forward in time according to a specified linkage function to return current and past state and outputs derived from that state. A forward query answers the question: *What did this state influence and how was that influence propagated?*

4.8.1 State queries

Arnold can recover past file versions, transient process state, inputs, and output. If a specified file version does not already exist in its cache, Arnold uses cooperative replay to regenerate the contents of the file. Arnold reexecutes the specified replay group to regenerate both transient process state and output. Inputs (with the exception of file data) are logged, so no reexecution is needed to retrieve them.

Arnold also provides an interactive state query to allow users to inspect GUI output. During replay, X server output is displayed on the current screen, allowing the user to observe the display as it was manipulated during recording. Via a separate console, the user can fast forward to a particular output (i.e., the display is updated at replay speed without the original think time, I/O delays, etc.) or pause the replay at a particular point in the execution. By delaying a specified amount after each X output, Arnold can also display a slow-motion execution.

4.8.2 Backward lineage queries

A backward query starts from a specified piece of current or past state. The first step in processing this query is to translate the starting state into a set of <replay group,system call,byte range> tuples, where the system call is a specific call executed by a process in the replay group, and the byte offset is a range of bytes returned by that call. There are many possible types of starting state. For instance, the starting state may be data in a current file specified as a <filename,starting offset,length> tuple. Arnold looks up the lineage of these bytes in the current filemap, which contains the <replay group,system call,byte offset> tuple of the system call that produced each byte.

The starting state may also be process inputs (e.g., data read from a socket, pipe, or terminal). The user can specify a replay group and a specific type of input (e.g., network data) or a specific input channel (e.g., the terminal device), as well as a regular expression over the inputs of that type or sent over the channel. In this case, Arnold uses the user-interface tools described in Section 4.7 to return a set of <replay group,system call,byte range> tuples that match the specification.

The starting state may also be process outputs, which are specified in a similar fashion and translated to tuples via replay of the group and application of the user-interface tools described in Section 4.7. Past file versions can be used as starting state by either specifying the output or input of the file data.

For X output, the user may use the GUI replay described previously to regenerate past GUI output, pause the replay, and click on the display to specify an <x,y> coordinate. Arnold identifies the most recent X output generated by the group at an area surrounding the <x,y> coordinate. The starting state is the bytes passed to the system calls or library functions generating that output.

Finally, the starting state can be arbitrary process state in the replayed group. Custom state must be specified via a Pin tool. For instance, we describe a case study in Section 6.4.3 in which we determine what data was leaked by the Heartbleed vulnerability. The Pin tool inspects program state at the faulty instructions and determines whether any bytes were sent over the network incorrectly. If so, those bytes are specified as starting state for a backward lineage query to determine what specific data was leaked.

Given one or more of the above starting states, Arnold translates that state into a set of <replay group,system call,byte range> tuples. It executes a deterministic re-

play for each unique replay group in the set. If the user specifies a specific linkage function as part of the query, Arnold uses the Pin tool associated with that linkage. The linkage taints all inputs to that process group during replay by assigning a unique identifier for each <system call,byte> read from an input source. When an output matching one of the target tuples is generated, Arnold outputs the set of taint identifiers (if any) associated with the target bytes.

If the user declines to specify a linkage function, Arnold runs multiple replays, stating with the most restrictive linkage and proceeding to less restrictive ones as long as no target output is influenced by any input. Thus, if the copy linkage returns no values, the data linkage is run, then the index linkage, etc.

When inputs are found to influence target outputs, Arnold checks the source of those inputs. If the input is from an external source (e.g., bytes read from a network socket), lineage can be traced no further. If the input is from an IPC system call or file read, Arnold determines the replay groups that generated the data from the replay group log. It then recursively replays those groups to trace lineage back one more step. If the input is from a file, Arnold also reads from the log the system call and byte offsets that wrote the data. Otherwise, such information can be obtained by replaying the specified group.

Finally, if Arnold traces lineage back to a user input from the GUI or keyboard, it attempts to infer a linkage between the data entered by the user and any information recently seen by the user. It searches through the output of applications currently displaying output at the time the input was entered using the user-interface tool of Section 4.7. It performs a fuzzy string matching to determine whether the output likely influenced the input. If a match is found, it reports this as a *human linkage* and continues to trace the lineage back from the system calls that generated the matching output. Because human linkages are imperfect and may report both false positives and false negatives, Arnold treats human linkages as the weakest form of linkage, giving them the lowest priority with respect to matching inputs to outputs.

In summary, a backward query proceeds in a tree-like search, fanning out from one or more target states. The search continues until it is stopped by the user or all state has been traced back to external system inputs. As the search fans out, Arnold replays multiple replay groups in parallel. In addition, if no lineage is specified, it may test multiple linkages for the same group in parallel, terminating less restrictive searches if a more restrictive search finds a linkage.

4.8.3 Forward Queries

Forward queries start from some past state and return the values influenced by that state according to a specified linkage function. As with backward queries, the starting state may consists of current or past file state, process inputs or outputs, and/or specific bytes in an address space identified by a Pin tool.

Tracing intra-process lineage for forward queries is much more efficient than for backward queries because only those bytes corresponding to the target input need to be tainted. This results in much less overhead in tracking information flow within each process.

The lineage tool returns a set of outputs that are influenced by the target inputs. Arnold uses the reverse index described in Section 4.5 to determine which replay groups subsequently consumed that output. Thus, the query fans out recursively from the initial state to replay the groups that were influenced by that data. As with backward queries, human linkages from user-visible output to user-generated input are also traced as part of the query. Further, as the query fans out, multiple replay groups are reexecuted in parallel.

The forward query returns the set of all current state and system outputs influenced by the starting state. It also contains the specific chain (processes executed, inputs, and outputs) that propagated that influence.

5 Privacy

One concern with eidetic systems is the potential exposure of private data. Arnold's log can be used to recreate any state on the system and thus must be protected to the same or greater degree that one would protect other private memory and file system state.

A related concern is preserving a user's ability to exclude data from recording. For example, Arnold as described here would preserve all state from a user's "private" browsing session. However, Arnold could provide the ability to remove sensitive portions of the process graph by sanitizing and replacing portions of the log. For instance, Arnold could remove sensitive information contained in a process's address space by replacing that process's execution with a simple stub program that produces the same output.

A user who wants to remove sensitive information may actually benefit from Arnold's ability to track lineage. The user could identify the sensitive information or portions of execution, then ask Arnold to identify points in the process graph that depend upon this information. The user could then replace those parts of the graph with stubs that jump over the sensitive portions. Of course, once data is removed from Arnold, all lineage information and ability to query the execution are also lost.

6 Evaluation

Our evaluation answers the following questions:

User	Days	Groups per day	Storage utilization (MB) per day			
			RAW file cache	Logs	Filemap	Total
A	25	995	475	267	36	779
B	24	475	1095	936	339	2064
C	21	26122	869	350	690	1910
D	16	3339	1675	82	838	2594

Table 1: Storage utilization during multi-week trial

- What is Arnold's performance overhead?
- What are Arnold's storage requirements?
- What types of queries can Arnold answer?

6.1 Storage overhead

We first consider the storage overhead of running an eidetic system. To measure this overhead, the authors of the paper continuously recorded activity on their workstations for several weeks. The recorded activity included all user-level processes started from a terminal or launched from the GUI. It did not include several system-level processes, such as the X server (which is not recorded per the design in Section 4.2), processes that directly manipulate the replay data (e.g., indexing tools), and the sshd server (since we sometimes needed to log in without replay enabled for testing and maintenance). With the above exceptions, the vast majority of user activity was recorded. No files were evicted from the copy-on-RAW file cache since storage utilization was reasonable.

Table 1 summarizes the storage cost of our eidetic system. The third column shows the average number of replay groups created per day; note that there may be many threads and processes within a single group. The number of groups created varies widely depending on workload; some users have a few long-running applications; others have workflows (e.g., compilation) that create many short-lived groups. The next columns show average storage utilization per day, broken down to show the individual storage utilization of the copy-on-RAW file cache, the replay log storage, and the filemap. Process checkpoints are included in the total but not shown separately, since they account for less than 2 MB per day of storage. Executables and shared libraries referenced by checkpoints are included in the copy-on-RAW file cache.

While we cannot make comprehensive claims about storage utilization without a wider user study, this preliminary data is very encouraging. All users require less than 2.6 GB of storage per day; a 4 TB drive would suffice for 4–14 years.

6.2 Benefits of compression

Next, we quantify the benefits of Arnold's storage optimizations. Table 2 shows results for several workloads. The `Firefox` workload measures a half-hour browsing session consisting of 15 minutes of active browsing across 8 complex Web sites (e.g., Facebook and stack overflow), followed by 15 minutes of idle usage with Gmail windows open (triggering periodic JavaScript execution). The `evince`, `gedit`, `gpaint`, `LibreOffice` spreadsheet, and `LibreOffice` presentation workloads use those applications intensely for 3 minutes. The `latex` workload builds a prior OSDI paper, and the `make` workload builds the libelf-0.8.9 library. In these and subsequent experiments, we ensure repeatable results by automating all GUI workloads with the Linux Desktop Testing Project library [1], which captures and emulates GUI events.

The first row in the table shows storage usage when all nondeterministic inputs are logged without any compression. The subsequent rows show the effect of cumulatively applying model-based compression, copy-on-RAW file caching, the deterministic X proxy, semi-deterministic time, and gzip compression. The final row shows the compression ratio compared to baseline due to all of Arnold's optimizations.

Arnold averages a 411:1 compression ratio compared to the baseline. For comparison, simply applying gzip to the baseline averages only a 6:1 compression ratio. At 36:1, `LibreOffice` presentation sees the lowest compression ratio; this is due to the recording of temporary files written and read by the application. In the future, Arnold could omit such data since the files are never read by other replay groups and Arnold could regenerate the file contents during replay.

6.3 Performance overhead

Next, we measure Arnold's performance overhead during recording (i.e., the overhead that would normally be experienced by a user). All tests were run on a computer running Ubuntu 12.04LTS with an 8 core Xeon E5620 2.4 GHz processor, 6 GB memory, and a 1 TB 7200 RPM hard drive. We measured several terminal and GUI applications, and one server workload (apache):

- **kernel copy** – `cp -a` of the 3.5.0 Linux source.
- **cvs checkout** – check out Arnold's kernel source (589 MB, 52730 files) from a repository accessible via a local, 1 Gb local network connection.
- **make** – compile the libelf-0.8.9 library.
- **latex** – build a prior OSDI paper with `latex/bibtex`.
- **apache** – run the apache benchmark on an apache 2.2.22 server, configured with mpm_prefork with 256 workers, and a client connected via a 1 Gb local network connection (5000 requests for a 34 KB page with 50 concurrent requests at a time.)
- **gedit** – open a 15,000 line C file and find/replace on a commonly occurring string.

	Storage utilization (MB)							
	Firefox	evince	gedit	gpaint	spreadsheet	presentation	latex	make
Baseline	4517.63	194.51	764.34	95.29	4362.49	455.10	20.05	86.69
Model-based compression	308.93	7.07	12.50	36.12	41.16	31.07	7.19	81.52
Copy-on-RAW file cache	283.37	2.63	8.41	34.77	22.63	23.83	0.25	6.13
X compression	173.74	2.61	8.29	1.08	22.55	15.94	0.25	6.13
Semi-deterministic time	127.72	2.11	6.51	0.94	19.23	15.39	0.29	6.12
Gzip	24.87	0.11	0.50	0.08	3.33	12.71	0.05	1.15
Compression ratio	182:1	1752:1	1530:1	1217:1	1311:1	36:1	393:1	75:1

Table 2: Reduction in storage utilization via incremental application of optimizations

- **facebook** – load the White House's public Facebook page in `Firefox` version 23.0 (the completion time is measured by the onLoad event.)
- **spreadsheet** – Open a 704 KB csv spreadsheet in `LibreOffice` 3.5 and convert it to an xml document.

Figure 1 shows the performance of Arnold on a variety of desktop workloads, normalized to the performance of an uninstrumented system. The middle bar for each workload shows the performance during recording; this is the overhead the user will experience during normal operation. The third bar shows the performance during recording when Arnold uses a second hard drive for logging, which minimizes interference with normal file system writes.

Arnold's overall performance impact is small: overhead is under 12% with a single disk for all but two workloads. The cvs checkout has approximately 100% overhead with a single disk because it saves all checked-out data twice: once as nondeterministic network input and again when cvs writes the data to the file system. Adding a second logging disk reduces the overhead for cvs to 15%.

The higher overhead seen by kernel copy is caused by saving filemap entries. This workload is disk-bound and creates many small files. For each file created, Arnold must create a B-tree to record lineage data—this is effectively a worst-case for saving filemap entries. A separate logging disk reduces the overhead to 1.7%.

The Facebook tests contained some outliers due to external network and servers. We eliminated gross outliers (500% or more above the median) from our measurements (both for baseline and for Arnold); doing so did not help Arnold disproportionately.

We also evaluate the scalability of Arnold on several Splash 2 benchmarks, shown in Figure 2. While scalability was not a focus of our work, Arnold has low overhead for all these benchmarks up to 8 threads. We attribute this to two factors. First, Arnold requires programs to be race-free, so it only has to check and log inter-thread synchronization operations rather than all shared-memory operations. Second, Arnold's model-based compression reduces the instrumentation overhead per synchronization

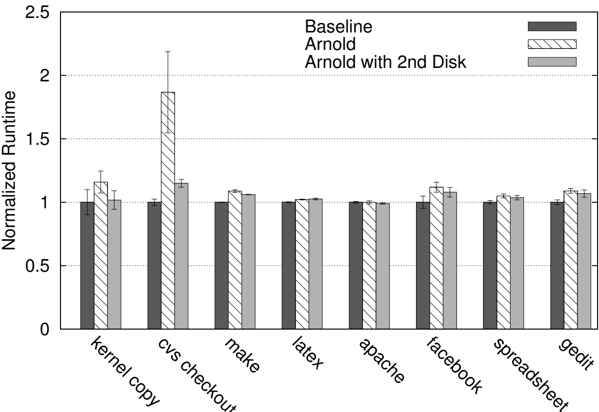


Figure 1: Arnold performance overhead normalized to unmodified Linux. Error bars are 95% confidence intervals.

operation to only two atomic instructions in the common case.

In summary, Arnold adds modest overheads of less than 12% with a single disk on all but 2 workloads over a wide range of desktop and interactive applications. Adding a second hard drive reduces the overhead to under 8% on all but one workload. In practice, even on single hard drive configurations, we noticed virtually no difference between our recorded applications and non-recorded applications. In fact, we needed to add a utility to our shell interface simply to determine whether recording was currently enabled or disabled.

6.4 Case studies

Finally, we look at a series of case studies of queries that we expect to be typical of Arnold's usage.

6.4.1 Backward Query

Our first case study is a typical backward query. In this scenario, a colleague points out to the user that he has cited the wrong paper in a conference submission. The user runs a backward query to determine how the incorrect citation was produced and what the correct citation should be.

We executed this query by opening a binary document with `xdvi`, scrolling to the bibliography, and clicking on a screen location to specify the incorrect citation as the

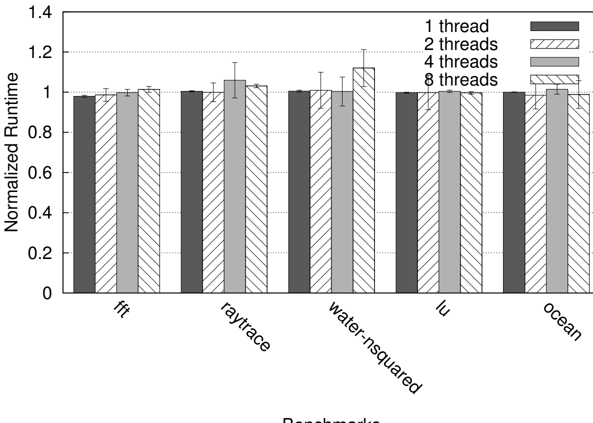


Figure 2: Arnold's scaling, normalized to unmodified Linux, on Splash2 benchmarks. Error bars are 95% confidence intervals.

starting state for the query. We did not specify a linkage, so Arnold ran the query with multiple linkage functions (the various linkage functions are analyzed in parallel via concurrent replay). For each step, Arnold chooses the most restrictive linkage function that produces some result (shown below in parentheses).

The query returned the following results:

- The specified output of `xdvi` came from the input file "paper.dvi" (index linkage).
- The incorrect citation in "paper.dvi" was generated by `latex` with data coming from the input file "paper.bbl" (data linkage).
- The data in "paper.bbl" was generated by `bibtex` with data from "full.bib" (copy linkage).
- The data in "full.bib" was generated by `vim` with data from the terminal device (copy linkage).
- A human linkage (described in section 4.7) reveals a fuzzy substring match between data coming from the terminal and `Firefox` output.
- The output displayed by `Firefox` came from a conference Web site (data linkage).
- The query also reports four false positives: a `latex` format file, a font file, `libc.so` and `libXt.so`.

Using Arnold, the user fast-forwards a `Firefox` replay to the point indicated by the query result. On viewing the recreated GUI, he realizes that the paper that he meant to cite was the *next* paper in the session after the incorrect citation.

As shown in the first row in Table 3, the query takes 209 seconds to execute, whereas the cumulative execution time of the recorded processes was only 96 seconds. Replay of the processes with zero instrumentation takes only 2 seconds because all user think time and most I/O delays are eliminated or replaced with a sequential disk read from the log. Simply attaching Pin to the replayed processes and inserting a very minimal instrumentation tool (which counts the number of system calls executed) increases the replay time to 70 seconds—this is the lower bound for any Pin tool on this workload.

The time it takes Arnold to perform the queries in table 3 is dominated by the Pin tool instrumentation and analysis, and not the actual replay system. Consequently, Arnold's query times are dictated by the number of instructions analyzed and the amount of taint information in the address space.

This query demonstrates that Arnold can successfully follow a long chain of applications to trace the lineage of data back to external inputs. The chain contains both binary and text data, as well as several types of linkages (intra-process, file, and human). Note that simply searching over inputs and outputs cannot reveal this whole chain (e.g., incoming Web data is encrypted, text input to `vim` includes backspaces and various keyboard macros, etc.) Lineage queries, however, can uncover linkages to such inputs because they directly observe the transformation of bytes in the process address space.

6.4.2 Forward Query

The second case study is a typical forward query. Our user now wishes to determine what other data and output has been affected by the faulty citation.

We executed this query by specifying the starting state as the incorrect citation in "full.bib." We also specified the index linkage function.

The forward query returns a list of external outputs and current files that the incorrect citation affected. Some key points of the result are:

- All subsequent versions of "full.bib" contain the incorrect citation. This is a shared bibliography file that is used to generate citations in several other papers on the user's computer. The forward query tracks the incorrect citation through the entire paper compilation process (e.g., though `bibtex`, `latex`, `dvips`, and `ps2pdf`).
- The query flags all files produced during the paper compilation process that include the specified citation (e.g., "paper.bbl", "paper.dvi", "paper.ps", and "paper.pdf.")
- The query does not return false positives. The user also has several papers that use the bibliography file "full.bib", but those papers do not cite the incorrect citation. Even though "full.bib" is read when those papers are compiled, Arnold correctly reports that no output file is affected by the incorrect citation.
- The query shows that the user had copied and pasted the incorrect citation from "full.bib" to another file, "paper.bib", using `vim`. The query also returns subsequent compilations and output files of

Case Study	Record Time	Replay Time	Replay & Pin	Query Time
Case Study 1: Backward Query	96.1s	2.2s	70.0s	209.5s
Case Study 2: Forward Query	30.3s	1.6s	80.4s	110.7s
Case Study 3: Forward Heartbleed Query	114.1s	0.1s	6.9s	19.7s
Case Study 3: Backward Heartbleed Query	230.3s	0.4s	139.5s	235.1s

Table 3: Summary of case studies

those compilations that reference the incorrect citation in "paper.bib".

- The query detects that the user ran a python script to produce a file with more succinct version of the citations, "small.bib", from "full.bib". It detects the incorrect citation in "small.bib" and in paper compilations that reference the incorrect citation from that file.
- The query detects that the user e-mailed a paper with the incorrect citation. This shows up as a network output of `sendmail`.
- The query returned no false positives.

The second row of Table 3 shows that the forward query required 111 seconds to execute, whereas simply replaying all processes with the simple Pin tool require 80 seconds. Thus, the relative overhead of the forward query instrumentation, is (as expected) much less than that for a backward query.

6.4.3 Heartbleed

Our third case study is motivated by the 2014 Heartbleed attack. One reason this attack caused such concern is that service providers were unable to determine what (if any) data was leaked. We show how Arnold is able to help an administrator determine whether sensitive data was leaked from a low-volume Web server, which hosts and stores a key-value database.

First, the administrator runs a forward query to see if the server's private key was leaked. This query requires a custom definition of output, so she creates a Pin tool. Heartbleed exploited a missing bounds check, so the Pin tool simply emulates the missing bounds check when the target instruction is reached and flags as output any data in excess of what the bounds check would have rejected. Her forward query specifies a starting state of the private key file, an output definition of only those bytes returned by the Pin tool, and the index linkage function.

We emulated this scenario by recording 100 GET and POST requests to Nginx 1.4.7 with OpenSSL version 1.0.1f (run times scale roughly linearly with the number of requests). This query took 20 seconds to perform and returned no outputs, showing that the private key was not leaked). We confirmed the correctness of this result manually.

Next, the administrator asks: *was any data leaked, and if so what data?* We constructed a backward query to answer that question. We used the custom Pin tool to define as starting state any data incorrectly sent due to the Heartbleed exploit and specified the index linkage. The backward query determined that:

- The Web server, `Nginx`, serviced a heartbeat request that leaked process memory.
- The leaked bytes came from a UNIX socket written by `FastCGI`, which is responsible for dynamic Web content.
- `FastCGI` received these bytes from a pipe written by a python script that it spawned.
- The script read the bytes from a database file.
- The bytes read from the database file came from POST requests that inserted those key-value pairs. This is determined by following the bytes back through a python script, `FastCGI`, and `Nginx`.

In summary, Arnold reveals both the leaked content *and* the origin of that data. Further queries could reveal the specific users who had their content leaked (e.g., by using a Pin tool to extract the userid from the connections that wrote the leaked data). The total query time was 235 seconds, roughly double the cost of replay and Pin instrumentation alone.

This case study shows the value of not limiting a priori the types of lineage that an eidetic system can track. For example, prior tools for intrusion recovery focus on inter-process lineage but cannot track intra-process lineage [23, 18, 22]. Upon learning of the vulnerability, the user can write new tools that detect data flows she had not anticipated at the time the system was being recorded, then use these tools on executions that were recorded before the tools existed.

7 Conclusions and future work

Arnold is a prototype of an eidetic system, targeted at personal computers and workstations. Arnold can recall any past user-level state, and it can trace the lineage of any byte in a current or prior state. This paper shows that the overheads of providing such functionality are reasonable: our results shows that adding a commodity hard drive can satisfy 4 or more years of storage needs with most runtime overheads under 8%. We have made the source code for Arnold available at https://github.com/endplay/omniplay.

Our case studies show the power of an eidetic system by recovering past state and tracing the lineage of data through a wide variety of applications and user interactions. The precision of combining operating system tracing of inter-process information flow with retrospective analysis of intra-process information flow yields accurate and informative query results.

We plan to explore several new aspects of eidetic systems. First, while Arnold can track all computation on a single machine, it cannot connect computation between multiple machines. One direction for future work is how to efficiently achieve Arnold's lineage tracking within large distributed systems. A second direction of future work is how to enable users to retroactively remove data from Arnold's log in order to preserve privacy. A complementary direction of future work is how to prevent or detect tampering with Arnold's log [20].

8 Acknowledgments

We thank the reviewers and our shepherd, Lorenzo Alvisi, for their thoughtful comments. This material is based upon work supported by the National Science Foundation under grants number CNS-0905149 and CNS-1017148. Yihe Huang created data race tools that we adapted for this work. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Linux Desktop Testing Project. `http://ldtp.freedesktop.org`.

[2] ADAMS, I. F., LONG, D. D. E., MILLER, E. L., PASUPATHY, S., AND STORER, M. W. Maximizing efficiency by trading storage for computation. In *Proceedings of the Workshop on Hot Topics in Cloud Computing* (June 2009).

[3] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 193–206.

[4] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation* (Hollywood, CA, October 2012).

[5] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[6] BACON, D. F., AND GOLDSTEIN, S. C. Hardware assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging* (1991), ACM Press, pp. 194–206.

[7] BENT, J., THAIN, D., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIVNY, M. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation* (March 2004).

[8] BERGER, E. D., YANG, T., LIU, T., AND NOVARK, G. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications* (Orlando, FL, October 2009), pp. 81–96.

[9] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel Java. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications* (Orlando, FL, October 2009), pp. 97–116.

[10] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 80–107.

[11] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference* (June 2008), pp. 1–14.

[12] CUI, H., WU, J., GALLAGHER, J., GUO, H., AND YANG, J. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 337–351.

[13] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2009), pp. 85–96.

[14] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[15] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M., AND CHEN, P. M. Execution replay on multiprocessor virtual machines. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2008), pp. 121–130.

[16] ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (September 2002), 375–408.

[17] FELDMAN, S. I., AND BROWN, C. B. IGOR: A system for program debugging via reversible execution. In

PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (1988), pp. 112–123.

[18] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The Taser intrusion recovery system. In *Proceedings of the 2005 Symposium on Operating Systems Principles* (October 2005).

[19] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[20] HAEBERLEN, A., ADITYA, P., RODRIGUES, R., AND DRUSCHEL, P. Accountable virtual machines. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[21] HOWER, D. R., AND HILL, M. D. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th International Symposium on Computer Architecture* (June 2008), pp. 265–276.

[22] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[23] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 223–236.

[24] LAADAN, O., BARATTO, R., PHUNG, D., POTTER, S., AND NIEH, J. DejaView: A personal virtual computer recorder. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, Oct 2007), pp. 279–292.

[25] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)* (June 2010), pp. 155–166.

[26] LEE, D., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Chimera: Hybrid program analysis for determinism. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation* (Beijing, China, June 2012).

[27] LEFEBVRE, G., CULLY, B., HEAD, C., SPEAR, M., HUTCHINSON, N., FEELEY, M., AND WARFIELD, A. Execution Mining. In *Proceedings of the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2012).

[28] LIU, T., CURTSINGER, C., AND BERGER, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 327–336.

[29] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[30] MONTESINOS, P., CEZE, L., AND TORRELLAS, J. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture* (June 2008), pp. 289–300.

[31] MUNISWAMY-REDDY, K.-K., AND HOLLAND, D. A. Causality-based versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (San Francisco, CA, February 2009).

[32] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, May/June 2006), pp. 43–56.

[33] NARAYANASAMY, S., PEREIRA, C., AND CALDER, B. Recording shared memory dependencies using Strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 229–240.

[34] NETZER, R. H. B. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* (1993), pp. 1–11.

[35] NEWSOME, J., AND SONG, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium* (February 2005).

[36] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, March 2008), pp. 308–318.

[37] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2009), pp. 97–108.

[38] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 177–191.

[39] PETERSON, Z. N. J., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transacations on Storage 1*, 2 (2005), 190–212.

[40] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the 1995 Winter USENIX Conference* (January 1995), pp. 213–223.

[41] POZNIANSKY, E., AND SCHUSTER, A. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPOPP03* (San Diego, CA, June 2003), pp. 179–190.

[42] RONSSE, M., AND DE BOSSCHERE, K. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems 17*, 2 (May 1999), 133–152.

[43] RUSSINOVICH, M., AND COGSWELL, B. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (1996), pp. 258–266.

[44] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. *SIGOPS Operating Systems Review 33*, 5 (1999), 110–123.

[45] SEREBRYANY, K., AND ISKHODZHANOV, T. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (December 2009).

[46] SRINIVASAN, S., ANDREWS, C., KANDULA, S., AND ZHOU, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference* (Boston, MA, June 2004), pp. 29–44.

[47] VAHDAT, A., AND ANDERSON, T. Transparent result caching. In *Proceedings of the 1998 USENIX Annual Technical Conference* (June 1998).

[48] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, October 2011).

[49] VEERARAGHAVAN, K., FLINN, J., NIGHTINGALE, E. B., AND NOBLE, B. quFiles: The right file at the right time. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (San Jose, CA, February 2010), pp. 1–14.

[50] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Long Beach, CA, March 2011).

[51] VLACHOS, E., GOODSTEIN, M. L., KOZUCH, M. A., CHEN, S., FALSAFI, B., GIBBONS, P. B., AND MOWRY, T. C. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, March 2010), pp. 271–284.

[52] WEERATUNGE, D., ZHANG, X., AND JAGANNATHAN, S. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2010), pp. 155–166.

[53] XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th International Symposium on Computer Architecture* (June 2003), pp. 122–135.

[54] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *MOBS07* (June 2007).

[55] ZAMFIR, C., AND CANDEA, G. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th ACM European Conference on Computer Systems* (April 2010), pp. 321–334.

Detecting Covert Timing Channels with Time-Deterministic Replay

Ang Chen
University of Pennsylvania

W. Brad Moore
Georgetown University

Hanjun Xiao
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Linh Thi Xuan Phan
University of Pennsylvania

Micah Sherr
Georgetown University

Wenchao Zhou
Georgetown University

Abstract

This paper presents a mechanism called *time-deterministic replay (TDR)* that can reproduce the execution of a program, *including its precise timing*. Without TDR, reproducing the timing of an execution is difficult because there are many sources of timing variability – such as preemptions, hardware interrupts, cache effects, scheduling decisions, etc. TDR uses a combination of techniques to either mitigate or eliminate most of these sources of variability. Using a prototype implementation of TDR in a Java Virtual Machine, we show that it is possible to reproduce the timing to within 1.85% of the original execution, even on commodity hardware.

The paper discusses several potential applications of TDR, and studies one of them in detail: the detection of a covert timing channel. Timing channels can be used to exfiltrate information from a compromised machine; they work by subtly varying the timing of the machine's outputs, and it is this variation that can be detected with TDR. Unlike prior solutions, which generally look for a specific type of timing channel, our approach can detect a wide variety of channels with high accuracy.

1 Introduction

When running software on a remote machine, it is common for users to care not only about the correctness of the results, but also about the time at which they arrive. Suppose, for instance, that Bob is a customer of a cloud computing platform that is run by Alice, and suppose Alice offers several machine types with different speeds, for which she charges different prices. If Bob chooses one of the faster machines to run his software but finds that the results arrive later than expected, he might wish to verify whether he is getting the service he is paying for. Conversely, if an angry Bob calls Alice's service hotline to complain, Alice might wish to convince Bob that he is in fact getting the promised service, and that the low performance is due to Bob's software.

A closely related problem has been studied in computer security. Suppose Charlie is a system administrator, and suppose one of his machines has been compromised by an adversary who wants to exfiltrate some data from the machine without raising Charlie's suspicion. In this case, the adversary might create a covert timing channel [31]—that is, he might cause the machine to subtly vary the timing of the network messages it sends, based on the data it is supposed to leak. As in the previous scenario, the outputs of the machine (in this case, the transmitted messages) are perfectly correct; the problem can only be detected by looking at the *timing*.

Although the two problems appear very different at first, they are in fact both instances of a more fundamental problem: *checking whether the timing of a sequence of outputs from a machine M is consistent with an execution of some particular software S on M*. The difference is in the part of the system that is being questioned. In the first scenario, it is the machine *M*: Bob suspects that Alice has given him a slower machine than the one he is paying for. In the second scenario, it is the software *S*: Charlie suspects that the adversary may have tampered with the software to vary the timing of the outputs. Thus, a solution for the underlying problem could benefit both of the scenarios we have motivated.

One possible approach would be to try to infer the "correct" timing of running the software *S* on the machine *M*, e.g., by carefully analyzing the timing of the various subroutines of *S*. But there are many factors that can affect the timing of a program's execution – cache effects, hardware interrupts, inputs at runtime, preemptions by the kernel or by other programs, I/O latencies, and many more – and their combined effect is extremely difficult to predict. Even inferring an upper bound can be very difficult, as the extensive literature on worst-case execution time (WCET) analysis [50] in the real-time systems domain can attest – and even an excellent WCET would still not be sufficient to solve our problem because we would need to know the specific runtime, not just an upper bound.

In this paper, we explore an alternative approach to this problem. Our key insight is that it is not necessary to *predict* the timing of S on M in advance – it is sufficient to *reproduce* the timing after the fact. If Bob had access to another machine M' of the same type and could reproduce the precise timing of S on that machine, he could simply compare the timing of the outputs during the reproduced execution to the timing of the messages he actually observed. If M was of the correct type and was indeed running S, the two should be identical; if they are not, either M must have had a different type, or S must have been modified or compromised. The potential advantage of this approach is that *there is no need to analytically understand the complex timing behavior of, e.g., caches or interrupt handlers*: if the two executions unfold in exactly the same way, the cache contents during the executions should be very similar as well.

Deterministic replay [19] provides a partial solution in that it can reproduce the *functional* behavior of a program by carefully recording all nondeterministic events (such as external inputs or random decisions) in a log, and by replaying the exact same events during replay. This ensures that the program produces the same outputs in the same order. However, it is not sufficient to reproduce the program's *temporal* behavior: as we will show experimentally, the replayed execution can take substantially more – or less – time than the original execution, and the outputs can appear at very different points in both executions. There are two key reasons for this. First, existing replay systems reproduce only factors that control a program's control or data flow; they do not reproduce factors that affect timing because the latter is not necessary for functional replay. Second, and more fundamentally, play and replay involve very different operations (e.g., writing vs. reading, and capturing vs. injecting) that have different timing behavior, and these differences affect the program's overall timing.

We propose to address these challenges using a mechanism we call *time-deterministic replay (TDR)*. A TDR system naturally provides deterministic replay, but it additionally reproduces events that have nondeterministic *timing*, and it carefully aligns its own operations during play and replay so that they affect the program's timing in a similar way. On an ideal TDR implementation, replay would take exactly the same time as play, but in practice, TDR is limited by the presence of *time noise* on the platform on which it runs: for instance, many CPUs speculatively execute instructions, and we do not know a way to reproduce this behavior exactly. Nevertheless, we show that it is possible to mitigate or eliminate many large sources of time noise, and that the timing of complex programs can be reliably reproduced on a commodity machine with an error of 1.85% or less.

We also describe the design of *Sanity*[1], a practical TDR system for the Java Virtual Machine, and we show how Sanity can be used in one of our target applications: the detection of covert timing channels. Detecting such channels is known to be a hard problem that has been studied for years. The best known solutions [15, 22, 23, 40] work by inspecting some high-level statistic (such as the entropy) of the network traffic, and by looking for specific patterns; thus, it is not difficult for an adversary to circumvent them by varying the timing in a slightly different way. To our knowledge, TDR offers the first truly *general* approach: it can in principle detect timing modifications to even a single packet, and it does not require prior knowledge of the encoding that the adversary will use. In summary, we make the following five contributions:

- The TDR concept itself (Section 2);
- The design of Sanity, a system that provides TDR for the Java Virtual Machine (Section 3);
- A prototype implementation of Sanity (Section 4);
- An application of TDR to the detection of covert timing channels (Section 5); and
- An experimental evaluation of Sanity (Section 6).

2 Overview

In this section, we give a more precise problem statement, and we explain some of the key challenges.

2.1 Problem statement

Figure 1 illustrates two variants of the scenario we consider in this paper. In the variant in Figure 1(a), Alice has promised Bob that she would run some software S on a (virtual or physical) machine of type T; Bob can connect to S over the network, but he does not have physical access to it. (This scenario commonly occurs on today's cloud platforms.) If the performance of S does not meet Bob's expectations, Bob might wonder whether Alice has really provisioned a machine of type T, or perhaps a less powerful type T'.

Figure 1(b) shows a different variant in which Charlie runs S on a machine he controls directly. Even if S appears to be working normally, Charlie might wonder whether the machine has been compromised by a remote adversary who has altered S and is now trying to leak secrets over the network by subtly altering the timing of the messages S sends. The key difference to the first scenario is that the machine is *known* to be of type T (perhaps Charlie can physically inspect it), and that Charlie is questioning the integrity of S instead.

[1]The name Sanity is a play on the definition of *insanity*, often attributed to Albert Einstein, as the exact repetition of a process while expecting a different outcome.

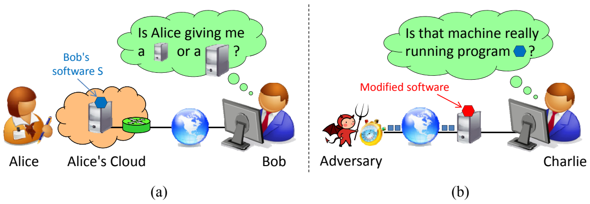


Figure 1: Two scenarios that benefit from TDR: *(a)* Bob wishes to verify that his software is running on the expected type of machine in Alice's cloud, and *(b)* Charlie wishes to verify that his machine is correctly executing his program.

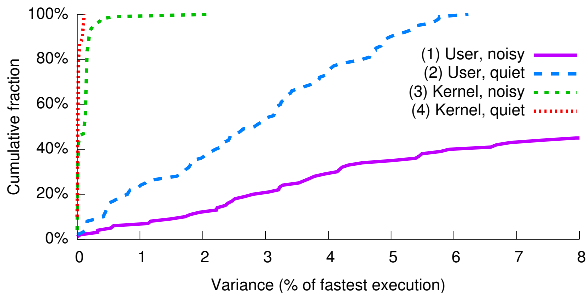


Figure 2: Timing variance of a simple program that zeroes out a 4 MB array, in four different scenarios.

In both scenarios, it seems reasonable to assume that Bob and Charlie have a way to observe the messages $m_1, m_2, \ldots$ that S sends and receives, as well as the approximate transmission or arrival time $t_1, t_2, \ldots$ of each message. The problem, then, is to *decide whether a sequence (m_i, t_i) of messages and message timings is consistent with an execution of a software S on a machine of type T.*

2.2 Why not use prediction?

If Bob and Charlie had a way to *precisely predict* how long an execution of S on T was going to take, they could solve the problem simply by comparing the observed message timings t_i to his predictions. However, the execution time depends on an enormous number of factors, such as the inputs the program receives, the state of the CPU's caches and TLBs, the number of hardware interrupts, the scheduling decisions of the kernel, and the duration of I/O operations, to name just a few. Because of this, predicting the execution time of any nontrivial program is known to be extremely difficult.

This problem has been extensively studied in the real-time systems community, and a variety of powerful timing analyses are now available (cf. [7] and [50] for an overview). But these analyses typically produce bounds – the *worst-case execution time (WCET)* and the *best-case execution time (BCET)* – and not the exact execution time. Moreover, the WCET (BCET) is typically much higher (lower) than the actual execution time [49]. Such bounds are useful if the goal is to guarantee timeliness, i.e., the execution completes before a particular point in time; however, it is usually not precise enough for the problem we consider here.

2.3 Approach: Reproducible timing

The solution we present in this paper is based on the key insight that it is not actually necessary to predict a priori how long an execution of S on T is going to take – *it would be sufficient to reproduce the timing of an execution after the fact*. This is a much easier problem because it does not require an analysis of the many complex factors that could affect the execution time; we "merely" need to ensure that these factors affect the reproduced execution in the same way. For instance, to *predict* the impact of the caches on the execution time, it would be necessary to predict the exact sequence of memory accesses that S is going to perform; to *reproduce* the impact, we can simply reproduce the same sequence of memory accesses.

Reproducible timing would solve the problem from Section 2.1 because it would enable a form of auditing: if Bob and Charlie locally have another machine of type T available, they can reproduce the execution of S on that machine and compare the timing to the timing they have observed on the remote machine. If the two are similar, this suggests that the remote machine is indeed of type T; if the timings are dissimilar, this is evidence that the machine is of a different type, or that it is not running the unmodified software S.

2.4 Challenge #1: Time noise

To highlight some of the key challenges of reproducible timing, we first consider two simple strawman solutions. The first is to simply reproduce the remote execution using deterministic replay, i.e., to ask the remote machine to record all nondeterministic inputs (such as messages, interrupts, etc.) along with the precise point in the program where each of them occurred, and to inject these inputs at the same points during replay. Deterministic replay is a well-established technique for which several mature solutions exist [3, 13, 19, 20, 39, 52, 53].

However, although this approach faithfully reproduces the control flow and the outputs of the original execution, it usually does *not* reproduce the timing. To illustrate this, we performed a simple experiment in which we measured the time it took to zero out a 4 MB array. Figure 2 shows a CDF of the completion times, normalized to the fastest time we observed. We show results for four different scenarios: (1) user level with GUI and network turned on; (2) user level in single-

user mode, running from a RAM disk; (3) kernel mode; and (4) kernel mode with IRQ turned off, cache flushed, TLB flushed, and execution pinned to a specific core. Ideally, all the executions would take the same amount of time and thus have a variance of zero, but in practice, some take considerably more time than others – the largest variance we observed was 189% in scenario (1), which corresponds to nearly 3x the time of the fastest execution in that scenario. Because of this variability, which we will refer to as *time noise* in the rest of this paper, it is extremely difficult to compare the timing of different executions, even for very simple programs.

However, Figure 2 also contains a hopeful message: as the environment becomes more and more controlled, the timing becomes more and more consistent. Hence, a major focus of this paper is on identifying and removing sources of time noise. If there were a way to completely eliminate all sources, the timing of the original and the repeated execution would be identical.

Where does the time noise come from? Commodity hardware and software have not been designed with repeatable timing in mind, and therefore contain many sources of time noise, including:

- **Memory:** Different memory accesses during play and replay and/or different memory layouts can increase or decrease the number of cache misses at all levels, and/or affect their timing;
- **CPU:** The processor can speculatively execute instructions or prefetch data, e.g., based on branch target predictions;
- **I/O:** Input/output operations can take a variable amount of time, particularly when HDDs are involved (due to seek/rotational latency);
- **IRQs:** Interrupts can occur at different points in the program; the handlers can cause delays and displace part of the working set from the cache; and
- **Kernel/VMM:** The kernel can preempt the program to schedule other tasks, or to take care of internal housekeeping. Also, system calls can take a variable amount of time.

Some of these sources can be completely eliminated; others can at least be considerably reduced by carefully designing the kernel or VMM. For instance, we can eliminate the variability from the address space layout by giving the program the same physical frames during play and replay, and we can reduce the interference from hardware interrupts by enabling them only at certain points in the execution.

2.5 Challenge #2: Play/replay asymmetry

Even if the timing of the program and the underlying hardware were completely deterministic, there is still a need to record its inputs so that its execution can be reproduced on another machine. This gives rise to another challenge: record and replay are fundamentally different operations that typically involve different code, different I/O operations, and different memory accesses. Thus, they will typically affect the timing in a different way.

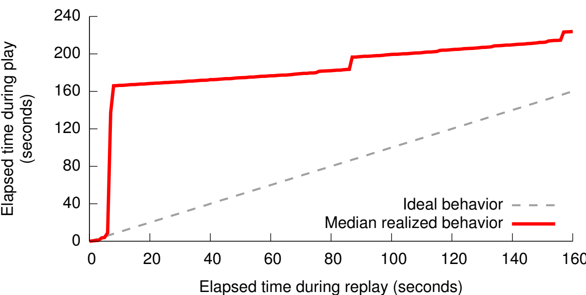


Figure 3: In existing VMMs, the timing during replay can differ substantially from the timing during play.

To illustrate this point, we performed the following simple experiment. We recorded the execution of a simple Linux VM using XenTT [13], a replay-enabled variant of the Xen hypervisor, and we directed some web requests towards an HTTP server that was running in the VM. To get a sense of the timing of the events in the VM, we instrumented XenTT to also record, for each event e, the wall-clock time $T_p(e)$. We then replayed the log and measured, for each event e, the wall-clock time $T_r(e)$ at which e was replayed. By comparing T_p and T_r, we can get a sense of the relative speed of play and replay.

Figure 3 shows a plot in which T_p is on the vertical axis and T_r on the horizontal axis; each point represents one event in the log. With fully time-deterministic replay, this graph would show a straight line, but the actual graph is far from it. There are some phases in which replay is faster than play, e.g., the interval from $T_p(e) = 183$ to $T_p(e) = 196$, in which the VMM was waiting for inputs; XenTT simply skips this phase during replay. In other phases, play is faster than replay, e.g., during the kernel boot phase, when Linux calibrates its internal clock.

This simple experiment shows that, to achieve repeatable timing, removing sources of time noise is not enough – the VMM also needs to "balance" play and replay in such a way that they affect the timing in approximately the same way.

3 Sanity Design

In this section, we describe the design of Sanity, a virtual-machine monitor (VMM) that provides highly repeatable timing. Sanity is a clean-slate VMM design that implements the Java Virtual Machine (JVM).

Noise source	Mitigation technique(s) used	Effect	Section
Divergence	Deterministic replay [19]	Eliminated	3.2
Randomness	Avoid or log random decisions	Eliminated	3.2
Scheduler	Deterministic multithreading [38]	Eliminated	3.2
Interrupts	Handle interrupts on a separate core	Reduced	3.3+3.4
Play vs. replay	Align JVM's control flow and memory accesses during play and replay	Eliminated	3.5
Caches	Flush caches at the beginning; use the same physical frames	Reduced	3.6
Paging	All memory is pinned and managed by JVM	Eliminated	3.6
I/O	Pad variable-time operations; use SSDs instead of HDDs	Reduced	3.7
Preemption	Run in kernel mode; do not share core with other apps	Eliminated	4.2
CPU features	Disable timing-relevant CPU features, such as frequency scaling	Reduced	4.2

Table 1: Sources of time noise that Sanity mitigates or eliminates, as well as the corresponding techniques.

3.1 Why a clean-slate design?

It is natural to ask why we have chosen to redesign Sanity from scratch instead of simply extending one of the many excellent open-source VMM implementations that are already available. The reason is that existing implementations were not built with time-determinism in mind and thus tend to contain a variety of sources of time noise, such as randomized data structures, system calls, various types of speculation, and so on. Finding these sources would be difficult because their effects are not necessarily local: for instance, a function A might invoke a system call, and in the process of handling it, the kernel might access different memory locations, depending on its current state; this might then add cache misses to the execution of a completely different and unrelated function B that runs several milliseconds later.

By building our VMM from scratch, we gained the ability to control every aspect of its design, and to carefully avoid introducing time noise at each step along the way. Since our resources were limited, we chose the Java Virtual Machine (JVM), which is relatively simple – it has only 202 instructions, no interrupts, and does not include legacy features like the x86 string instructions – and for which there is a large amount of existing software. However, even state-of-the-art JVMs are very complex; for instance, the HotSpot JVM consists of almost 250,000 lines of code. Hence, we necessarily had to focus on the core features and omit others, such as just-in-time (JIT) compilation, which obviously limits Sanity's performance. We accept this limitation because it is not inherent: given enough time and a large enough team, it should be perfectly feasible to build a time-deterministic JIT compiler, as well as all the other features we were unable to include in our prototype.

Sanity provides deterministic replay (Section 3.2), and it includes a combination of several techniques that reduce or eliminate time noise (Sections 3.3–3.7). Table 1 provides an overview of the sources of time noise we focused on, and the technique(s) we used to mitigate or eliminate each of them.

3.2 Deterministic replay

Our implementation of deterministic replay in Sanity relies mostly on standard techniques from other replay-enabled JVM implementations [2, 16]: during the original execution ("play"), we record all nondeterministic events in a log, and during the reproduced execution ("replay"), we inject the same events at the same points. For the JVM, this is much easier than for the x86-based replay implementations that many readers will be familiar with (e.g., ReVirt [19]). This is because the latter must record asynchronous events, such as hardware interrupts, that can strike at any point during the execution – even in the middle of CISC instructions such as `rep movsb` – which requires complex logic for injecting the event at exactly the same point during replay. The JVM, in contrast, does not have a notion of interrupts, and a simple global instruction counter is sufficient to identify any point in the execution.

To reduce the number of events that must be recorded, we implement a simple form of deterministic multithreading [38] in Sanity: threads are scheduled round-robin, and each runnable thread is given a fixed budget of Java instructions it may execute before it is forced to yield. Since the execution of the individual threads is already deterministic, there is no need to record information about context switches in the log, since they will occur at exactly the same points during replay.

If Sanity is used for long-running services – perhaps a web server, which can run for months or even years – it is important to enable auditors to reproduce smaller segments of the execution individually. Like other deterministic replay systems, Sanity could provide checkpointing for this purpose, and thus enable the auditor to replay any segment that starts at a checkpoint.

3.3 Timed core and supporting core

Although the JVM itself does not have asynchronous events, the platform on which it runs (e.g., an x86 machine) will usually have them. To prevent these events from interfering with the timing of the JVM's execu-

tion, Sanity confines them to a separate core. Thus, even though Sanity implements a single-core JVM, it requires a platform with at least two cores: a *timed core (TC)* that executes the JVM itself, and a *supporting core (SC)* that handles interrupts and I/O on the TC's behalf.

The TC-SC separation shields the TC from most effects of asynchronous events, but, on most platforms (with the exception of certain NUMAs), it cannot shield it *entirely*, since the two cores share the same memory bus. Even if the SC's program fits entirely into the SC cache, DMAs from devices must still traverse the memory bus, where they can sometimes compete with the TC's accesses.

3.4 Communication between TC and SC

The TC and SC communicate by means of two in-memory ring buffers: the *S-T buffer* and the *T-S buffer*. The SC receives asynchronous inputs, such as incoming network messages, and writes them to the S-T buffer; the TC inspects this buffer at regular intervals, or when an explicit read operation is invoked (such as `DatagramChannel.read`). Conversely, when the TC produces outputs (e.g., outgoing network messages), it writes them to the T-S buffer. The SC periodically inspects this buffer and forwards any outputs it contains.

The purpose of this arrangement is to make play and replay look identical from the perspective of the TC – in both cases, the TC reads inputs from the S-T buffer and writes outputs to the T-S buffer. The SC, of course, acts differently during replay: it reads the inputs in the S-T buffer from the log rather than from a device, and it discards the outputs in the T-S buffer. But the control flow on the TC, which is crucial for repeatable timing, is identical. (See Section 3.5 for an important exception.)

3.5 Symmetric read/writes

If both the TC's sequence of memory accesses and its control flow are to be exactly the same during play and replay, there are two special cases that need to be handled. The first concerns the T-S buffer. Suppose, for instance, that the VM invokes `System.nanoTime` to read the current wallclock time. This event must be logged, along with the value that Sanity returns, so that the call can have the same result during replay. A naïve implementation might check a "replay flag" and then write the time to the T-S buffer if the flag is clear, and read from it if the flag is set. However, this would cause both different memory accesses (a dirty cache line during play, and a clean one during replay) and different control flow (perhaps a branch taken during play and not taken during replay, which would pollute the BTB).

To produce the exact same control flow and memory accesses during play and replay, we use the approach

```
void accessInt(int *value, int *buf) {
  int temp = (*value) & playMask;
  temp = temp | (*buf & ~playMask);
  *value = *buf = temp;
}
```

Figure 4: Algorithm for symmetric reads/writes.

shown in Figure 4 to access events in the T-S buffer. (The figure shows, as an example, how we access an integer.) `playMask` is a bit mask that is set to $111\ldots11$ during play, and to 0 during replay. When an event occurs, Sanity invokes the algorithm with `*value` set to the value that would need to be recorded if this were the play phase (e.g., the current wallclock time). The algorithm then reads from the T-S buffer the data `*buf` that would need to be returned if this were the replay phase. It then computes the value `temp` to be either the former (during play) or the latter (during replay). Finally, it writes `temp` to the T-S buffer *and* returns it to the caller; the caller then proceeds with the returned value (e.g., returns it from `System.nanoTime`). The overall effect is that the value is written to the buffer during play and read from the buffer during replay; the memory accesses are identical, and no branches are taken.

A related case concerns the S-T buffer. When the TC checks the buffer during play and finds a new entry there (e.g., a received network packet), it must write the JVM's instruction counter to the entry as a virtual "timestamp" so it can be injected at the same point during replay. During replay, the TC must check this timestamp to avoid processing entries before the instruction counter reaches that value again. We handle this case similarly to the first one (the TC *always* reads, checks, and writes the timestamp), but with an additional twist: during play, the SC always adds a "fake" entry with a timestamp of infinity at the end of the buffer, so that the TC's next-entry checks will always fail. When the SC appends a new entry, it overwrites the previous "fake" entry (but adds a new one at the end), and it sets the timestamp to zero, so the TC's check is guaranteed to succeed. The TC can recognize this special value and replaces it with the current instruction count.

3.6 Initialization and quiescence

To maximize the similarity between play and replay timing, Sanity must ensure that the machine is in the same state when the execution begins. This not only involves CPU state, but also memory contents, stable storage, and key devices.

On the TC, Sanity flushes all the caches it can control, including the data caches, the TLB, and any instruction caches. This entails a certain performance penalty be-

cause the caches must all be repopulated, but recall that the caches remain enabled during the execution, so it is a one-time cost. We note that some CPUs perform cache flushes asynchronously (such as the `wbinvd` instruction on IA-32). To account for this, the TC adds a brief *quiescence* period before it begins the execution; this allows the cache flush to complete, and it can also be used to let hardware devices (such as the NIC) finish any operations that are still in progress. If the instruction stream on the TC is exactly the same and the caches have a deterministic replacement policy (such as the popular LRU), this is *almost* sufficient to reproduce the evolution of cache states on the TC. The missing piece is virtual memory: even if the TC has the same virtual memory layout during play and replay, the pages could still be backed by different physical frames, which could lead to different conflicts in physically-indexed caches. To prevent this, Sanity deterministically chooses the frames that will be mapped to the TC's address space, so they are the same during play and replay.

During execution, no memory pages are allocated or released on the TC; the JVM performs its own memory management via garbage collection. Garbage collection is not a source of time noise, as long as it is itself deterministic.

3.7 I/O handling

Sanity uses the SC to perform all I/O operations. For streaming-type devices, such as the NIC or the terminal, this is relatively straightforward: whenever the TC has an output to send (such as a network packet, or a terminal output), it writes the data to the T-S buffer; whenever the SC receives an input, it writes it to the S-T buffer, which the TC checks at regular intervals.

Storage devices are more challenging because the latency between the point where the VM issues a read request and the point where the data is available can be difficult to reproduce. A common way to address this (cf. [5]) is to pad all requests to their maximal duration. This approach is expensive for HDDs because of their high rotational latency, which can be several milliseconds, but it is more practical for the increasingly common SSDs, which are roughly three orders of magnitude faster, and far more predictable.

3.8 What Sanity does *not* do

We emphasize that Sanity does *not* run with caches disabled, and that it does *not* prevent the JVM from performing I/O or from communicating with the outside world. Although the effects of these features on execution time are hard to *predict*, we argue – and we will demonstrate in Section 6 – that it is possible to *reproduce* them with relatively high accuracy, to a degree that becomes useful for interesting new applications (more about this in Section 5). Ensuring reproducibility is far from trivial, but can be accomplished with careful design choices, such as the ones we have described here.

4 Sanity Implementation

Next, we describe a prototype of Sanity that we have built for our experiments.

4.1 Java Virtual Machine

For our prototype, we implemented a Java Virtual Machine from the ground up. This includes support for the JVM's instructions, dynamic memory management, a mark-and-sweep garbage collector, class loading, exception handling, etc. However, we designed our JVM to be compatible with Oracle's Java class library (`rt.jar`), so we did not need to re-implement the standard classes in the `java.lang` package. The class library interacts with the JVM by calling native functions at certain points, e.g., to perform I/O. For our experiments, we implemented only a subset of these functions; for instance, we did not add support for a GUI because none of our example applications require one.

Our current prototype does *not* support just-in-time compilation or Java reflection. As discussed in Section 3.1, we decided against implementing these because both are major sources of complexity, and neither is likely to be a major source of time noise. Since Oracle's class library invokes reflection at some surprising points (e.g., when instantiating a `HashMap`), we made some small additions to the class library that can replace the relevant classes without using reflection.

Altogether, our prototype consists of 9,061 lines of C/C++ code; our additions to the class library contribute another 1,150 lines of Java code.

4.2 Isolating the timed core

Recall from Section 3.3 that the timed core must be isolated, to the extent possible, from sources of time noise in the rest of the system. One way to accomplish this would be to run the JVM as a standalone kernel; however, we decided against this because of the need for driver support. Instead, we implemented our prototype as a Linux kernel module with two threads. The TC thread runs on one core with interrupts and the NMI disabled; the SC thread runs on a different core and interacts with the TC as discussed in Section 3.4. The SC thread can access the kernel's device drivers, e.g., to send and receive network packets. On NUMA platforms, the two cores should be chosen to be far apart, so they share as little as possible.

To improve the repeatability of cache effects, our prototype uses the same physical memory frames for each execution. We use a separate kernel module for this purpose that is loaded during startup and that reserves a certain range of frames for later use by the TC/SC module.

To reduce the time noise from the CPU itself, we disable timing-relevant features such as frequency scaling and TurboBoost in the BIOS (and, in the case of the latter, again during boot, since Linux surreptitiously re-enables it). Disabling dynamic hardware-level optimizations has a certain performance cost, but it seems unavoidable, since the effect of these optimizations is unpredictable and – at least on current hardware – they cannot be fully controlled by the software. To further reduce the time noise from the CPU, we carefully flush all caches before the execution starts; specifically, we toggle `CR4.PCIDE` to flush all TLB entries (including global ones) and we use the `wbinvd` instruction to flush the caches.

4.3 Limitations

Since our Sanity prototype is built on commodity hardware and a commodity kernel, it cannot *guarantee* time-determinism, since we cannot rule out the possibility that there is a source of time noise that we have missed. It should be possible to achieve such a guarantee by enforcing time-determinism at each layer – e.g., by starting with a precision-timed system such as PRET [21] and by adding a kernel that is built using the principles from Section 3 – but this is beyond the scope of this paper.

Our Sanity design assumes that play and replay will be performed on machines of the same type. It may be possible to correct for small differences, e.g., by using frequency scaling during replay to match a lower clock speed during play, or by disabling extra cores or memory banks that were not available during play. However we are not aware of any efficient technique that could precisely reproduce the timing of an execution on a completely different architecture.

Two final limitations result from the fact that our design uses exactly two cores, one TC and one SC. First, the SC is mostly idle because its only purpose is to isolate the TC; thus, the second core is mostly overhead. Second, multithreaded Java programs must be executed entirely on the TC and cannot take advantage of additional cores. These are limitations of our Sanity prototype, and not of TDR: the TC/SC division, and thus the need for a second core, could be avoided in a TDR system that runs directly on the hardware, and the restriction to a single TC could be removed by adapting techniques from existing multi-core replay systems, such as SMP-ReVirt [20], perhaps in combination with novel hardware features, as in QuickRec [41].

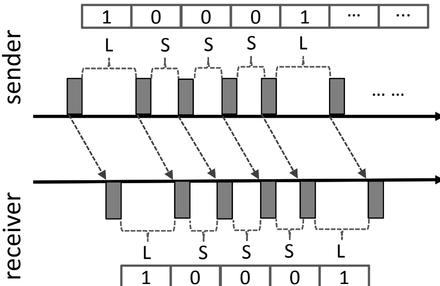


Figure 5: An example covert timing channel that encodes the bitstring `10001`.

5 Application: Covert Timing Channels

Next, we present a concrete example application for TDR: the detection of covert timing channels that exfiltrate information from a compromised machine.

A *covert channel* is an unauthorized channel that allows the surreptitious communication of information. Covert channels have become a pervasive security threat in distributed systems, and have produced an arms race between methods for achieving covertness and techniques for detecting such channels (see Section 8). Here, we focus on a class of covert channels called *covert timing channels* in which a compromised host manipulates the timing of network activities to directly embed covert information into inter-packet delays (IPDs). By observing the timing of packets, the receiver can reconstruct the covert message.

Figure 5 illustrates a simple covert timing channel. The sender manipulates the delays between sending two consecutive packets to encode a covert message, where bit `1` (resp. `0`) is encoded by adding a large (resp. small) IPD, indicated as 'L' (resp. 'S') in the Figure. Upon receiving the network flow, the receiver can then recover the covert message by observing the IPDs between consecutive packets.

5.1 Examples of timing channels

Since Lampson first proposed covert timing channels in the 1970s [31], a number of practical channels have been demonstrated in the literature (cf. [4, 11, 12, 14, 15, 45–47]), broadly falling into the following categories:

IP covert timing channel (IPCTC). Like most early timing channels, IPCTC is based on a simple idea: the sender transmits bit `1` by sending a packet within a predetermined time interval, and transmits `0` by remaining silent in that interval. Due to their unique traffic signatures, IPCTCs are straightforward to detect.

Traffic replay covert timing channel (TRCTC). TRCTC tries to confuse detectors by replaying the IPDs from legitimate traffic (without covert channels). It categorizes IPDs in the legitimate traffic stream into two

bins (B_0 and B_1 for small and large IPDs, respectively). It then transmits a 0 by choosing a delay from B_0 and a 1 by choosing a delay from B_1. However, since the encoding scheme is constant, TRCTC exhibits more regular patterns than a legitimate traffic stream, the latter of which usually has high variability (e.g., burstiness).

Model-based covert timing channel (MBCTC). MBCTC generates IPDs to mimic the statistical properties of legitimate traffic. It periodically fits samples of a legitimate traffic to several models and picks the best fit. Gianvecchio et al. [24] have shown that the shape of the MBCTC traffic is almost the same as the one of legitimate traffic. However, as there is no correlation between consecutive IPDs, MBCTC is highly regular.

There are multiple types of defenses against covert timing channels; for instance, it is possible to disrupt them or to reduce their bandwidth [5, 28, 30, 32], and certain types can even be eliminated with language-based techniques [1] (albeit at the expense of limiting expressiveness). However, recent work has begun to focus more on detection, typically based on traffic statistics. It is this work that is most relevant here.

5.2 State of the art in detection

Next, we briefly describe four state-of-the-art detection techniques for covert timing channels.

Shape test [15]. The shape test checks only flow-level statistics; it assumes that the covert channel traffic could be differentiated from legitimate traffic using only first-order statistics, such as the mean and variance of IPDs. While the shape test detects IPCTCs, it is often insufficient to detect more advanced timing channels.

Kolmogorov-Smirnov test (KS-test) [40]. The KS-test assumes that the legitimate traffic and the covert channel traffic are drawn from two different distributions. The KS-test calculates the distance between the empirical distributions of the test sample and training sample (from legitimate traffic). If the distance is above a predetermined threshold, the test distribution is considered to contain a covert timing channel.

Regularity test (RT-test) [15]. The RT-test is based on the observation that the variance of IPDs in legitimate traffic varies over time, while a covert channel manifests a relatively constant variance due to its constant encoding scheme. RT-test groups the traffic into sets of w packets, and compares the standard deviation of pairwise differences between each set. RT-test has the disadvantage that it exhibits a high false positive rate when the legitimate traffic is highly variant.

Corrected conditional entropy (CCE) [22]. The CCE metric extends the notion of the regularity test. It uses a high-order entropy rate to recognize the repeated pattern that is formed by the covert timing channel.

5.3 Detecting timing channels with TDR

Existing statistic-based detection techniques rely on the availability of a sufficient amount of legitimate traffic to construct accurate models and tune cut-off thresholds. This greatly reduces the effectiveness of these techniques when such information is not available. In addition, statistic-based techniques are effective when covert timing channels transmit information at a high rate; it is much more difficult to detect slow-rate covert timing channels in which traffic patterns are almost indistinguishable from legitimate ones.

To address these limitations, we propose a novel detection technique for covert timing channels that is based on TDR. Our approach differs fundamentally from the existing ones in Section 5.2 in that we do not look for the presence or absence of specific patterns in the observed traffic; rather, we use TDR to reconstruct what the timing of the packets ought to have been. Concretely, each machine would be required to record its inputs in a log; this log could then be audited periodically, perhaps authenticated with an accountability technique like PeerReview [25], and then replayed with TDR on a different machine, using a known-good implementation of the audited machine's software. In the absence of timing channels, the packet timing during replay should match any observations during play (e.g., from traffic traces); any significant deviation would be a strong sign that a channel is present.

Note that this approach does *not* require knowledge of a specific encoding, and that it can in principle detect even a delay of a *single* packet. The adversary's only way to avoid detection would be to make very small changes to the timing, so that they stay below TDR's replay accuracy; however, if the accuracy is high enough, the adversary may no longer be able to distinguish the timing changes from network jitter, which effectively renders the channel unusable.

6 Evaluation

Next, we report results from our experimental evaluation of Sanity. We focus on three key questions: 1) How accurately does Sanity reproduce the timing of the original execution?, 2) What are the costs of running Sanity?, and 3) Is Sanity effective in detecting a variety of covert timing channels?

6.1 Experimental setup

For our experiments, we deployed Sanity on a Dell Optiplex 9020 workstation, which has a 3.40 Ghz Intel Core i7-4770 CPU, 16 GB of RAM, an 128 GB Vector ZDO SSD, and a 1 Gbps network card. We installed Ubuntu

Benchmark	Sanity	Oracle-INT	Oracle-JIT
SOR	7.4211	1	0.2634
SMM	1.0674	1	1.1200
MC	4.0890	1	0.0305
FFT	8.4068	1	0.1590
LU	0.2555	1	0.0353

Table 2: SciMark2 performance of Sanity and Oracle's JVM, normalized to Oracle's JVM in interpreted mode.

13.12 as the host OS, and we configured it with a RAM disk for storing the logs and the files for the NFS server.

We also installed the 32-bit version of Oracle's Java SE 7u51 JVM, so we can compare the performance of Sanity to that of a state-of-the-art JVM. However, Oracle's JVM supports just-in-time (JIT) compilation, whereas Sanity does not; hence, to enable meaningful comparisons, we report two sets of results for the Oracle JVM: one with the default settings, and another in which the `-Xint` flag is set. The latter forces the JVM to interpret the Java bytecode rather than compiling it, just like Sanity. We refer to these configurations as Oracle-JIT and Oracle-INT, respectively.

6.2 Speed

The first question we examine is whether the presence of TDR imposes a major performance penalty. Ideally, we would simply "enable" and "disable" TDR in the same codebase, but this is hard to do because TDR is a design feature. However, we can at least obtain some qualitative results by comparing the results from a computation-intensive benchmark.

As a workload, we chose NIST's SciMark 2.0 [42] Java benchmark that consists of five computational kernels: a fast Fourier transform (FFT), a Jacobi successive over-relaxation (SOR), a Monte Carlo integration (MC), a sparse matrix multiply (SMM) and a lower-upper factorization (LU). We ran each benchmark in each of our three configurations (Sanity, Oracle-JIT, and Oracle-INT), and we measured the completion time.

Table 2 shows our results. Since Sanity lacks a JIT compiler, it cannot match the performance of Oracle's JVM with JIT compilation enabled. However, the comparison with the Oracle JVM in interpreted mode is more mixed; sometimes one JVM is faster, and sometimes the other. We note that Sanity has some advantages over the Oracle JVM, such as the second core and the privilege of running in kernel mode with pinned memory and IRQs disabled, so this is not a completely fair comparison. Nevertheless, at the very least, these results suggest that TDR is not impractical.

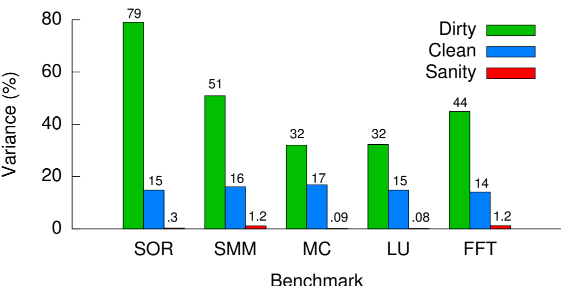


Figure 6: Timing variance for SciMark2, using either Sanity or the Oracle JVM in the "dirty" and "clean" configurations (see text).

6.3 Timing stability

A key requirement for any TDR implementation is timing *stability*: two executions of the same program with the same inputs and the same initial state must take a very similar amount of time. To quantify the stability of Sanity's timing, we again use the SciMark benchmark because it takes no external inputs, so it is easy to reproduce the same execution even without deterministic replay. We ran each benchmark on Sanity 50 times, and we calculated the difference between the longest and the shortest execution. For comparison, we performed the same experiment on two variants of the Oracle-INT configuration: a "dirty" variant, in which the machine is in multi-user mode, with a GUI and with networking enabled, and a "clean" variant in which the machine is in single-user mode and the JVM is the only program running. The latter approximates the closest one can get to timing stability with an out-of-the-box Oracle JVM.

Figure 6 shows our results. Not surprisingly, timing in the "dirty" configuration can vary considerably, in some cases by 79%; this is because of the many sources of time noise (such as preemptions and concurrent background tasks) that are present in this configuration. In the "clean" configuration, the variability is more than an order of magnitude lower; Sanity can reduce it by another order of magnitude or more, to the point where *all execution times are within 0.08%–1.22% of each other* (the corresponding bars in Figure 6 are there, but are hard to see). This suggests that Sanity effectively mitigates or eliminates the major sources of time noise.

6.4 Replay accuracy

Next, we examine how well Sanity can fulfill its promise of repeatable timing. For this purpose, we use an I/O-intensive benchmark because I/O is a major source of time noise; also, this allows us to easily collect many traces with different inputs and thus different timing behavior. We chose `nfsj` [37], an open-source NFS server that is written in Java. We made some small mod-

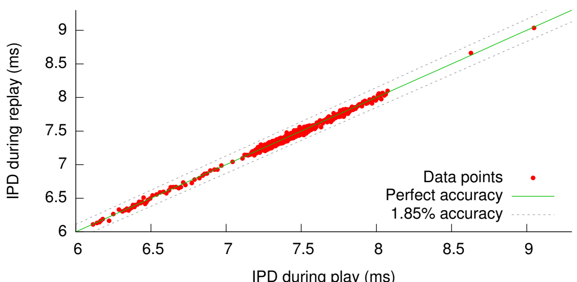


Figure 7: Comparison of inter-packet delays during play and replay, for the NFS traces.

ifications to `nfsj` to adapt it to our Java runtime, mostly to constructs that require reflection, which Sanity does not yet support.

We gathered 100 one-minute traces of the NFS server while it was handling requests, and we then replayed each of the traces. As a first sanity check, we compared the original execution time of each trace to the execution times of its replay. We found that 97% of the replays were within 1% of the original execution time; the largest difference we observed was 1.85%. Recall that deterministic replay requires very different operations in the VMM during play and replay, so this result is far more remarkable than the stability we have shown in Section 6.3: it is a result of the careful alignment of play and replay in Sanity's design.

To examine these results in more detail, we also recorded the timing of each individual message the NFS server transmitted during each play and replay. We then took each pair of consecutive messages (m^i_j, m^i_{j+1}) in each replayed trace T_i and calculated the difference between a) the transmission times of these messages during replay, and b) the transmission times of the corresponding messages during play, shown respectively on the y- and x-axes of Figure 7. If Sanity had reproduced the timing *exactly*, the two differences would be identical, and the graph would show a straight line; in practice, there is some remaining variability due to remaining sources of time noise. However, all the differences are within 1.85%.

6.5 Log size

An important source of overhead in Sanity is the log of nondeterministic events that the SC must write to stable storage during the execution. To quantify this, we examined our NFS traces from Section 6.4 and found that the logs grew at a rate of approximately 20 kB/minute. Not surprisingly, the logs mostly contained incoming network packets (84% in our trace); recall that these must be recorded in their entirety, so that they can be injected again during replay. (In contrast, packets that the NFS server transmits need not be recorded because the replayed execution will produce an exact copy.) A small fraction of the log consisted of other entries, e.g., entries that record the wall-clock time during play when the VM invokes `System.nanoTime`.

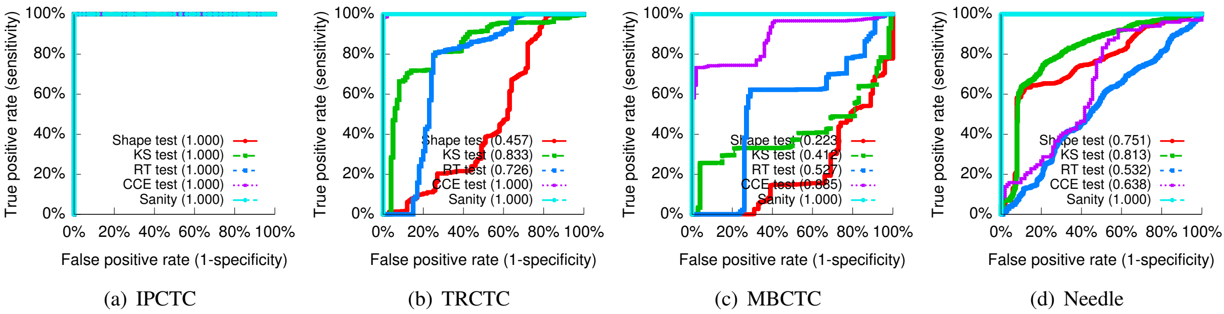
We note that Sanity requires no additional log entries specifically for TDR, so its logs should generally be no larger (or smaller) than those of previous implementations of deterministic replay. For instance, Dunlap et al. [19], which describes a replay system for IA-32, reported a log growth rate of 1.4 GB/day for SPECweb99, and 0.2 GB/day for a desktop machine in day-to-day use. We expect that Sanity's logs would have a similar size, so, given today's inexpensive storage, keeping the logs for a few days should be perfectly feasible.

6.6 Covert-channel experiments

In the rest of this section, we report results from our covert-channel experiments. For these experiments, we implemented the IPCTC, TRCTC, and MBCTC covert channels from Section 5.1 in our `nfsj`-based NFS file server. The channels add delays using a special JVM primitive that we can enable or disable at runtime; this allows us to easily collect traces with and without timing channels, without making changes to the server code.

In a real attack – e.g., in the cloud computing scenario – the server's messages would need to traverse a wide-area network and thus incur additional time noise that must be considered by the sender of a covert timing channel. To represent this in our experiments, we locate the NFS client and server at two different universities on the U.S. East coast. The RTT between the two was approximately 10 ms, and (based on 1000 ICMP `ping` measurements) the 50th, 90th, and 99th percentile jitter was 0.18 ms, 0.80 ms, and 3.91 ms, respectively. Since the content of the files on the NFS server is irrelevant, we simply used a workload of 30 files with sizes between 1kB and 30kB; the client reads all of these files one after the other.

To compare Sanity against the timing detectors described in Section 5.1 – shape test, KS-test, regularity test (RT-Test), and corrected conditional entropy (CCE-Test) – we ran experiments with each detector-channel combination. During each experiment, we gathered a packet trace on the server itself; this eliminates detection errors due to network jitter, and it approximates a scenario where the detector is very close to the server (for instance, it might be run by the network administrator). The traces were available to detectors; our Sanity-based detector additionally had access to the server's log and (for replay) to a non-compromised `nfsj` binary.



(a) IPCTC (b) TRCTC (c) MBCTC (d) Needle

Figure 8: ROC curves for our four covert channels and five detectors. Area under the curve (AUC) is shown in parentheses in the legends.

6.7 Detection performance: Haystacks

To make comparisons amongst the detectors, we vary the discrimination threshold of each detection technique. For the Sanity-based detector, this is the minimum difference between an observed IPD and the corresponding IPD during replay that will cause the detector to report the presence of a channel; the other detectors have similar thresholds. For each setting, we obtain a true-positive and a false-positive rate, and we plot these in a graph to obtain each detector's receiver operating characteristic (ROC) curve. An ideal detector would exhibit perfect recall (a true positive rate of 1.0) and specificity (a false positive rate of 0), and is depicted in a ROC curve as an upside-down L. We also measure the area under the curve (AUC) of each ROC curve, which correspondingly ranges from 0 (poor classification) to 1 (an ideal classifier).

Figures 8a–c show the resulting ROC curves for the IPCTC, TRCTC, and MBCTC channels. As expected, the simplistic IPCTC technique is detected by all tests, confirming earlier results [15]. The other channels more successfully evade detection when pitted against a mismatched detection technique; for instance, TRCTC does well against shape tests but is detectable by more advanced detection techniques; it preserves first-order traffic statistic but produces a distribution of IPDs that significantly differs from that of normal traffic. As expected, our Sanity-based detector offered perfect detection (AUC=1), which confirms that it can match or exceed the performance of existing detectors.

6.8 Detection performance: Needles

Next, we consider our fourth covert timing channel, which differs from all the other channels in that it is short-lived. This represents a more realistic scenario in which the sender (the NFS server) wishes to exfiltrate a small secret—for example, a password or private key. To minimize the risk of detection, the sender toggles its use of the covert channel, transmitting a single bit once every 100 packets. Thus, the channel does not change high-level traffic statistics very much, which makes it very difficult to detect with existing methods.

Figure 8d shows the ROC curves for this channel. As expected, all the existing detectors failed to reliably detect the channel; in contrast, our Sanity-based detector still provided perfect accuracy. This is expected because, unlike existing detectors, Sanity does not rely on statistical tests that must be carefully tuned to balance the risks of under- and over-detection; instead, TDR-based detectors can potentially find *any* significant timing variation, even if it affects only a single packet.

6.9 Time noise vs. jitter

As discussed in Section 3 and empirically measured in Sections 6.3 and 6.4, TDR does not completely eliminate all sources of time noise. For example, contention between the SC and the TC on the memory bus might affect different executions in slightly different ways. In theory, an adversary could exploit this remaining time noise to construct a covert channel that avoids detection: if the differences between log and replay due to the covert channel are within the range of normal time noise, then Sanity will fail to detect the channel.

However, such a strategy is infeasible in practice due to the vast asymmetry between time noise allowed by Sanity and time noise due to the network (i.e., network jitter). Figure 7 demonstrated that the timing noise allowed by Sanity is under 1.85% of the original IPDs, that is, 0.14 ms for a median IPD of 7.4 ms. On the other hand, the measured median jitter is 0.18 ms, which is 129% of the allowed noise. Note that the jitter is measured between two well-provisioned universities; it is a very conservative estimation of the jitter that the traffic of a covert timing channel would encounter. For example, the median jitter of broadband connection has been measured to be approximately 2.5 ms [18]. To avoid detection, the adversary would need to accept an extremely low accuracy of reception, making such an avoidance strategy impractical.

6.10 Summary

Our results confirm that it is possible to reproduce execution times with high accuracy, even on commodity hardware. Our prototype currently cannot match the performance of a state-of-the-art JVM, but, as discussed in Section 3.1 it should be possible to get better performance by adding features such as JIT. As an example of an interesting new application that TDR enables, we have demonstrated a novel, TDR-based detector for covert timing channels that outperforms existing detectors both in terms of accuracy and in terms of generality. To avoid triggering this detector, an adversary would need to use extremely small timing variations that would most likely be lost in the jitter of a wide-area network.

7 Discussion

Multi-tenancy: Although Sanity currently supports only a single VM per machine, it should be possible to provide TDR on machines that are running multiple VMs. The key challenge would be isolation: the extra VMs would introduce additional time noise into each other's execution, e.g., via the shared memory bus. We speculate that recent work in the real-time domain [51] could mitigate the "cross-talk" between different VMs; techniques such as [33] could be used to partition the memory and the cache. If the partitions are equivalent, it may even be possible to replay an execution in a different partition from the one in which it was recorded, which would remove the need to have the same physical pages available during play and replay.

Accountability: Although TDR can *detect* inconsistencies between the timing of messages and the machine configuration that supposedly produced them, it cannot directly *prove* such inconsistencies to a third party. This capability could be added by combining TDR with accountability techniques, such as accountable virtual machines [26]. However, the latter are designed for asynchronous systems, so a key challenge would be to extend them with some form of "evidence of time".

8 Related Work

Covert timing channels. We have already discussed prior work on timing channel detection in Sections 5.1 and 5.2. Briefly, TDR is different in two ways: 1) it looks at the timing of individual packets rather than high-level traffic statistics, which gives it the ability to detect even low-rate, sporadic channels, and 2) it does not look for specific anomalies but rather for deviations from the expected timing, which enables the detection of novel channels. We know of only one other work, Liu et al. [34] that uses a VM-based detector, but [34] simply replicates incoming traffic to two VMs on the *same* machine and compares the timing of the outputs. Moreover, without determinism the two VMs would soon diverge and cause a large number of false positives.

Deterministic replay. There is a large body of work on enabling deterministic replay at various levels, e.g., at the hardware level [27] or the OS level [3, 6, 8–10, 17, 36, 38, 39]. However, these solutions reproduce only the *functional* behavior of the program. To our knowledge, TDR is the first primitive that can accurately reproduce the *temporal* behavior as well.

Real-time systems. Timing stability has also been a design goal of precision-timed (PRET) machines [21]. The PRET machines reduce variances in the execution time using deadline instructions [29], thread-interleaved pipeline [35], and scratchpad-based memory hierarchy [35, 43]. These machines can potentially achieve a very stable timing, although they do require new processor designs. There also exist time-predictable architectures for real-time systems that can indirectly enable stable timing, such as MCGREP [48] and JOP [44], by making the execution time more deterministic. However, we are not aware of any existing work that provides both repeatable timing and deterministic replay.

9 Conclusion

This paper introduces time-deterministic replay (TDR), a mechanism for reproducing both the execution and the timing of a program. TDR is well-suited for a number of system administrator and developer tasks, including debugging, forensics, and attack detection.

TDR presents a number of design and engineering challenges—modern commodity processors and operating systems are tailored for performance, not for precise and identical repetition of processes. We eliminate or mitigate many sources of "time noise" and demonstrate the feasibility of TDR by implementing a proof-of-concept TDR-capable JVM that we call Sanity. Our benchmarking experiments reveal that Sanity can reproduce the timing of an execution to within 1.85% of the original. We additionally demonstrate the practicality of TDR by using Sanity to detect a variety of classes of covert timing channels. Our results are encouraging: Sanity is able to detect even extremely subtle and low-bandwidth timing channels that fail to be detected using standard shape- and statistical-based detection approaches.

Acknowledgments: We thank our shepherd Peter Chen and the anonymous reviewers for their comments and suggestions. This work was supported by NSF grants CNS-1065130, CNS-1054229, CNS-1149832, CNS-1064986, CNS-1117185, and CNS-1040672, as well as DARPA contract FA8650-11-C-7189.

References

[1] J. Agat. Transforming out timing leaks. In *Proc. POPL*, Jan. 2000.

[2] B. Alpern, T. Ngo, J.-D. Choi, and M. Sridharan. DejaVu: Deterministic Java Replay Debugger for Jalapeño Java Virtual Machine. In *OOPSLA Addendum*, Oct. 2000.

[3] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proc. SOSP*, Oct. 2009.

[4] S. Arimoto. An algorithm for computing the capacity of arbitrary discrete memoryless channels. *IEEE Trans. Inf. Theor.*, 18 (1):14–20, Sept. 2006.

[5] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proc. CCS*, Oct. 2010.

[6] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proc. OSDI*, Oct. 2010.

[7] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM TECS*, 13(4):82:1–37, 2014.

[8] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proc. ASPLOS*, Mar. 2010.

[9] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proc. OSDI*, Oct. 2010.

[10] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Proc. OOPSLA*, Oct. 2009.

[11] V. Berk, A. Giani, and G. Cybenko. Detection of covert channel encoding in network packet delays. Technical Report TR2005-536, Dartmouth College.

[12] R. Blahut. Computation of channel capacity and rate-distortion functions. *IEEE Trans. Inf. Theor.*, 18(4):460–473, 1972.

[13] A. Burtsev. Xen deterministic time-travel (XenTT). `http://www.cs.utah.edu/~aburtsev/xen-tt-doc/`.

[14] S. Cabuk. *Network Covert Channels: Design, Analysis, Detection, and Elimination*. PhD thesis, Purdue Univ., Dec. 2006.

[15] S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: Design and detection. In *Proc. CCS*, Oct. 2004.

[16] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proc. SPDT*, Aug. 1998.

[17] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *Proc. ASPLOS*, Mar. 2009.

[18] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing Residential Broadband Networks. In *Proc. IMC*, Oct. 2007.

[19] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI*, Dec. 2002.

[20] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay for multiprocessor virtual machines. In *Proc. VEE*, Mar. 2008.

[21] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Proc. DAC*, June 2007.

[22] S. Gianvecchio and H. Wang. Detecting covert timing channels: An entropy-based approach. In *Proc. CCS*, Oct. 2007.

[23] S. Gianvecchio and H. Wang. An Entropy-Based Approach to Detecting Covert Timing Channels. *IEEE Transactions on Dependable and Secure Computing*, 8(6):785–797, Nov 2011.

[24] S. Gianvecchio, H. Wang, D. Wijesekera, and S. Jajodia. Model-based covert timing channels: Automated modeling and evasion. In *Proc. RAID*, Sept. 2008.

[25] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, Oct. 2007.

[26] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proc. OSDI*, Oct. 2010.

[27] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *Proc. HPCA*, 2011.

[28] W.-M. Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium on Security and Privacy*, May 1991.

[29] N. J. H. Ip and S. A. Edwards. A processor extension for cycle-accurate real-time software. In *Proc. EUC*, Aug. 2006.

[30] M. H. Kang, I. S. Moskowitz, and D. C. Lee. A network pump. *IEEE Trans. Softw. Eng.*, 22:329–338, May 1996.

[31] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, Oct. 1973.

[32] P. Li, D. Gao, and M. K. Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In *Proc. DSN*, June 2013.

[33] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proc. RTAS*, June 1997.

[34] A. Liu, J. Chen, and H. Wechsler. Real-time covert timing channel detection in networked virtual environments. In *Proc. International Conference on Digital Forensics*. Jan. 2013.

[35] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proc. ICCD*, Sept. 2012.

[36] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *Proc. SOSP*, Oct. 2011.

[37] nfsj. `https://code.google.com/p/nfsj/`.

[38] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proc. ASPLOS*, Mar. 2009.

[39] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proc. SOSP*, Oct. 2009.

[40] P. Peng, P. Ning, and D. Reeves. On the secrecy of timing-based active watermarking trace-back techniques. In *Proc. IEEE Security and Privacy*, May 2006.

[41] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas. QuickRec: Prototyping an Intel architecture extension for record and replay of multithreaded programs. In *Proc. ISCA*, June 2013.

[42] R. Pozo and B. Miller. SciMark 2.0. `http://math.nist.gov/scimark2/`.

[43] J. Reineke, I. Liu, H. Patel, S. Kim, and E. A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *Proc. CODES+ISSS*, Oct. 2011.

[44] M. Schoeberl. A java processor architecture for embedded real-time systems. *J. Syst. Archit.*, 54(1-2):265–286, Jan. 2008.

[45] G. Shah, A. Molina, and M. Blaze. Keyboards and covert channels. In *Proc. USENIX Security*, July 2006.

[46] X. Wang and D. S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of inter-packet delays. In *Proc. CCS*, Oct. 2003.

[47] X. Wang, S. Chen, and S. Jajodia. Tracking anonymous peer-to-peer VoIP calls on the internet. In *Proc. CCS*, Nov. 2005.

[48] J. Whitham and N. Audsley. MCGREP–A predictable architecture for embedded real-time systems. In *Proc. RTSS*, Dec. 2006.

[49] R. Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, *Handbook on Embedded Syst.* CRC Press, 2005.

[50] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem–overview of methods and survey of tools. *ACM TECS*, 7(3):36:1–36:53, May 2008.

[51] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *Proc. RTSS*, Dec. 2013.

[52] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. MoBS*, June 2007.

[53] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: Object centRic DEterministic Replay for Java. In *Proc. USENIX ATC*, June 2011.

Identifying information disclosure in web applications with retroactive auditing

Haogang Chen, Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek
MIT CSAIL

Abstract

RAIL is a framework for building web applications that can precisely identify inappropriately disclosed data after a vulnerability is discovered. To do so, RAIL introduces *retroactive disclosure auditing*: re-running the application with previous inputs once the vulnerability is fixed, to determine what data *should* have been disclosed. A key challenge for RAIL is to reconcile state divergence between the original and replay executions, so that the differences between executions precisely correspond to inappropriately disclosed data. RAIL provides application developers with APIs to address this challenge, by identifying sensitive data, assigning semantic names to non-deterministic inputs, and tracking dependencies.

Results from a prototype of RAIL built on top of the Meteor framework show that RAIL can quickly and precisely identify data disclosure from complex attacks, including programming bugs, administrative mistakes, and stolen passwords. RAIL incurs up to 22% throughput overhead and 0.5 KB storage overhead per request. Porting three existing web applications required fewer than 25 lines of code changes per application.

1 Introduction

Unintentional disclosure of sensitive information is a common problem, despite improvements in security techniques and widespread use of best practices. Newspapers frequently report such leaks at companies, hospitals, universities, government institutions, etc. This paper is based on the premise that disclosures will remain common, since even if the best security mechanism and practices are used, humans will make mistakes: a programmer may introduce a bug, a user may choose a weak password, or a system administrator may misconfigure the access control policy. Even if a state-of-the-art security system is in place, human operators can still overlook alerts [13], inadvertently disclosing confidential data.

Dealing with data leaks can be expensive because institutions are often required by law to inform their users of the security breach. For example, the University of Maryland suffered a compromise and paid for one year of credit monitoring for 309,079 potentially affected users, since it was unable to immediately pinpoint which of the users were actually affected [15]. However, a subsequent manual audit, which took about a month, revealed that only a handful of users' information was disclosed, and that the bulk of the cost was unnecessary. This example is typical of the challenges administrators face after a leak.

The usual approach for identifying data disclosures is to maintain access logs and to analyze those logs after a security breach, in an attempt to identify who accessed what data, and to separate out the legitimate accesses from the illegal ones. Although there are challenges in maintaining access logs (see, for example, Keypad [6]), the hard problem is deciding whether data accesses were legitimate or not. Manually auditing all accesses is labor-intensive and imprecise, as illustrated by the University of Maryland example.

To reduce the cost of handling leaks, this paper explores a different, automated approach for deciding which accesses were legitimate or not, based on record and replay. In particular, the paper describes the design of a new system, named RAIL (Retroactive Auditing for Information Leakage), that can precisely identify whose information was leaked in the context of web applications, such as a health care application that collects patients' personal health information or a class submission web site for assignments and grades.

RAIL's main contribution is to apply record and replay to identifying improper disclosures. Record and replay has been used for many *integrity* applications, from analyzing attacks [9] to detecting past intrusions [8, 14] and recovering integrity [2, 3, 7], but prior work did not address the problem of dealing with past data disclosures. During regular operation, RAIL records sufficient information so that it can faithfully replay an application's requests later. Once a vulnerability has been identified, an administrator repairs the underlying cause in the application (e.g., fixing a bug in the application's source code, or changing an access control list), and then asks RAIL to replay requests. If RAIL notices a difference between data sent to users in the original run and the replay run, it will report that data as having been inappropriately disclosed. For example, if one user's account was compromised, RAIL will report only the portion of that user's data that was inappropriately accessed by an adversary.

Precisely detecting data disclosures using record and replay is challenging for several reasons. The core challenge is that the application may behave differently during replay due to non-determinism. For example, a homework submission system might randomly assign students to one another for code review. If during replay some of the stu-

dents are missing (e.g., because they were the attackers), the system might produce an entirely different assignment for code review. As a result, the replay will send different homework submissions to each student, and RAIL might report all previous homeworks as having been inappropriately disclosed. Previous record and replay systems do not have adequate solutions to this problem; they take a best-effort approach, and any final state is acceptable in the end, as long as all effects of the attack are gone [2, 7]. In contrast, RAIL's goal is to minimize divergence between normal execution and replay, in order to precisely identify illegal data disclosures.

A second challenge lies in identifying what represents a data item in the first place. For example, in the homework submission system, what is the unit of data disclosure that should be reported to the administrator?

A third challenge lies in tracking dependencies in application code at a fine granularity (e.g., individual functions). Previous systems either tracked code dependencies at a coarse granularity (e.g., source files loaded by the application [2]), or made extensive changes to the interpreter to record fine-grained dependencies [8]. However, neither approach is ideal in practice.

Finally, a fourth challenge is making replay fast so that an administrator can quickly audit for data disclosures over long periods of time. One month of requests must not take a month replay.

RAIL addresses these challenges by providing an explicit API for developers to help administrators record and replay applications. For instance, in the homework submission system, the programmer uses RAIL's API to assign semantic names to random pairings between students (see §7.2), enabling the system to preserve assignments during replay, even if some students are gone. The API includes annotations to identify data, assign semantic names to non-deterministic inputs, and record dependencies on state for selective replay.

We implemented the RAIL API in the context of Meteor [11], a framework for building web applications. The API's design is not limited to Meteor. We chose Meteor because it cleanly separates data items and web interfaces via asynchronous messages. Because of this property (which is common in modern web frameworks), we were able to implement much of RAIL inside of Meteor, greatly reducing the need for application changes. In fact, we were able to port existing, deployed Meteor applications (e.g., a health survey application, a homework submission application, and a social news application) to RAIL with few changes to the application code.

We evaluated RAIL using these applications and several synthetic attacks, based on common vulnerabilities (e.g., code bugs and user mistakes) that result in direct data disclosures or back doors that leak data indirectly. Our results show that RAIL is precise, efficient, and practical: RAIL accurately flags all inappropriate disclosures with few false reports and minimal re-execution; the throughput and storage overhead of RAIL during recording is 22% and 0.5 KB per request, respectively; and porting several web applications to use RAIL's API required fewer than 25 lines of code changes per application.

RAIL cannot identify all data leaks. For example, attacks that copy the database from the server through some external mechanism (e.g., an NSA employee with access to the server) are outside of the scope of RAIL. In general, RAIL does *not* handle attacks by system administrators, or covert channels; RAIL focuses on data disclosed through the web application's normal interface.

The rest of the paper is organized as follows. §2 discusses previous related work. §3 shows how to use RAIL from the perspective of site administrators and application developers. §4 summarizes RAIL's assumptions and requirements. §5 describes the high-level design of RAIL. §6 presents RAIL's uniform interface for managing shared objects. §7 details the replay and handling of non-determinism. §8 describes our prototype implementation of RAIL. §9 evaluates RAIL's effectiveness. §10 discusses our experience of with RAIL. §11 concludes.

2 Related work

RAIL is the first practical system for precisely auditing unauthorized data disclosures. Much of the previous work on auditing has focused on logging *all* accesses to confidential data. For example, Keypad [6] and Pasture [10] use either cryptography or trusted hardware to maintain a centralized audit log of data accesses, while allowing low-overhead access to this data across many distributed devices. While this model is a good one for auditing unauthorized access when a user's device is stolen, it cannot distinguish legitimate from unauthorized accesses if there is a mistake in the access control policy.

Information flow control and taint tracking systems, such as TaintDroid [5] and TightLip [16], try to prevent disclosure of confidential data in the first place. However, we believe such systems cannot be 100% effective, and disclosures will still happen. For example, a system administrator may misconfigure labels, or a user's password may be guessed by an attacker. Unlike these systems, RAIL does not try to prevent any data leaks; rather, it can detect the leak after the fact without a priori knowledge about which data is sensitive. Moreover, although RAIL and TightLip [16] share the common idea of comparing execution outputs, RAIL addresses the unique challenge of reconciling state divergence, which improves auditing accuracy and performance.

Similarly, encryption is often used to prevent data disclosure in the face of a compromised server, such as in the Mylar web framework [12]. However, encryption does not protect against all disclosures, such as when an ad-

```
1  var Users = App.getDBCollection('users');
2  var Homeworks = App.getDBCollection('hws');
3  var Answers = App.getDBCollection('answers');
4  App.publish('pub_ans', function (userid) {
5 -  var uid = userid;
6 +  var uid = App.getSessionUserId();
7    var u = Users.findOne( {_id: uid} );
8    if (u && u.profile.type === 'staff')
9      return Answers.findAll();
10   return Answers.find( {user: uid} );
11 });
12 App.method('submit', function(hw_id, answer) {
13   var uid = App.getSessionUserId();
14   var hw = Homework.findOne( {_id: hw_id} );
15   var ctx = Rail.inputContext(hw_id, uid);
16   if (!uid || !hw || hw.dueDate < ctx.date())
17     throw new Error('Submission failed');
18   Answers.insert( {_id: ctx.random(),
19     hw: hw_id, user: uid, answer: answer} );
20 });
```

Figure 1: Part of the server-side code from a homework submission application as our running example. Invocations of built-in framework APIs are prefixed with `App`; RAIL-specific API are prefixed with `Rail`.

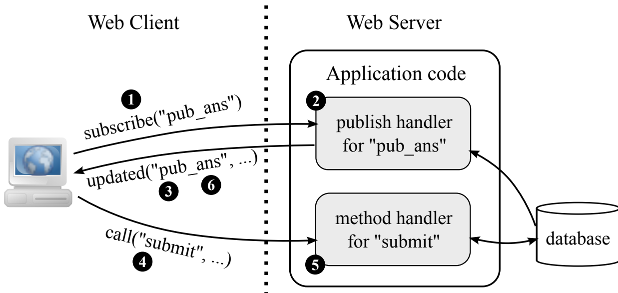


Figure 2: Typical workflow for the running example. 1) The client sends an RPC request to subscribe to the "`pub_ans`" dataset. 2) The corresponding publish handler is executed, which returns a query. 3) The server runs the query and sends the initial dataset to the client via "`updated`" messages. 4) The client calls the RPC method "`submit`" to hand in an answer to a homework. 5) The server runs the method handler, which updates the database. 6) The server reruns published queries affected by the update, and pushes updates to *all* clients that subscribed to it, via several "`updated`" messages.

ministrator misconfigures a system or when programmers make mistakes in the application logic.

RAIL uses many ideas from prior work on record and replay, such as the action history graph and selective replay from Retro [7], and the comparison of normal execution and replay from Rad [14] and Poirot [8]. RAIL's key contribution lies in providing an API that programmers can use to minimize divergence during replay. Prior replay systems were focused on restoring integrity, and reducing divergence was not a priority, resulting in heuristic solutions that attempted to match up replay with normal execution but did not offer strong guarantees. For example, Retro matches non-deterministic system calls in sequential order [7], which might not make sense in our homework submission system's code review assignments. Poirot [8] stops replay when it detects the entry point of an attack, and reports only the initial problematic request; RAIL identifies leaked data in all future requests that are indirectly affected by the attack, and reports precisely which data items were disclosed.

Brown's undoable mail server [1] proposes structuring server software around a *verb* API to handle replay after state changes. However, Brown's API is not designed to identify data items that may be disclosed.

3 Using RAIL

Using RAIL with an application involves three main phases. First, the application developer modifies their application's source code to invoke the RAIL API. Second, during normal operation, RAIL records inputs to the web application, along with other information specified through the RAIL API, to a log. Third, when an administrator detects that there was a problem, she can describe the problem to RAIL (e.g., supply a patch or fix an access control list, and pinpoint the time when the problem first arose). RAIL will replay requests from the start of the problem, detect which data items may have been inappropriately disclosed as a result, and report them to the administrator.

To understand how this works, consider an example application: a website for submitting homework assignments. Figure 1 shows the server-side code of this application, written in the Meteor framework [11], along with changes that the developer would make to use the RAIL API. Figure 2 illustrates a typical workflow for the code. The application defines an RPC method "`submit`" (line 12), which allows students to submit their answers to a homework. The framework does not explicitly send data to the clients, but adopts a publish–subscribe pattern: the server publishes a database query with a name (line 4); when the results of the query might change, the server reruns the query and pushes any updates to all clients that subscribed to it. As we can see from lines 7–10, the publish code returns different queries based on the user's account profile: course staff members are permitted to see all submissions, but students can see only their own.

Suppose the application developer made a mistake checking permissions; as can be seen on lines 5–6, the mistake allows the client to supply the current user ID as an argument to the `pub_ans` subscription, instead of using the `App.getSessionUserId` method, which returns the currently authenticated user ID; such a mistake was discovered in the Telescope social news application [4]. This mistake could have been exploited by an adversary to view all students' submissions, by supplying the user ID of a staff member when subscribing to `pub_ans`.

After running the application with RAIL for a while, the site administrator discovers the vulnerability. She wants to know if an adversary exploited the bug, and whose homework submissions were disclosed as a result. To do so, she first applies a patch that fixes the bug (lines 5–6),

```
Leaked data for session RuZw9cCaDMJLdsj8G:
Login: evil_student @ 4/24/2014 3:14:15 PM
   IP: 192.168.0.10
  - answers/fNKXudhNDF7  fields: answer, grade, ...
  - answers/jxT5w7jRJpm  fields: answer, grade, ...
...
```

Figure 3: An example report from Rail indicating that several homework submissions were inappropriately disclosed.

and specifies a time before any possible disclosures (e.g., the time of the first submission). Then she launches the web application again in replay mode. Rail re-executes all subsequent events which might be affected by the patch. Finally, Rail compares the new data items sent to each client with those from the original execution, and generates a disclosure report that details any differences. For example, Figure 3 shows a possible report for this example, indicating that several homework submissions were inappropriately disclosed to a client at a particular IP address.

In order to precisely identify disclosed data, Rail requires application developers to use Rail APIs to name and access shared objects and to annotate non-deterministic inputs in their code. These names, known as *context identifiers*, help Rail match up semantically equivalent operations between the original execution and re-execution. For example, on line 15 of Figure 1, the code creates an input context with an identifier composed of the homework ID and user ID, and uses the context to generate dates and random numbers (lines 16 and 18). As long as the identifier remains unchanged during re-execution, Rail will reproduce the same date and random number from the context. Rail also relies on context identifiers to track dependencies, as we describe in §6.

All non-deterministic inputs and shared objects, including current date and time, random numbers, session variables, database records, and top-level functions, must be accessed via a Rail wrapper to preserve access semantics during replay. In principle, this could be a burden for the programmer, but in our experience, most of the wrapping can be confined to the web framework itself, requiring little additional per-application effort from the developer. In the example application from Figure 1, the developer uses standard APIs from the underlying web framework to retrieve the currently logged-in user (lines 6 and 13), and access the database (lines 7, 9, 10, 14, and 18). Behind the scenes, the web framework itself contains calls to the Rail APIs that wrap these objects, taking care of object naming and dependency tracking.

4 Assumptions

Rail relies on the following assumptions to work properly. First, the developer should correctly use Rail APIs to access shared objects, read non-deterministic inputs, and generate outputs. Developers should also name context identifiers appropriately so that states can be matched up during re-execution.

Second, Rail assumes that the inputs from clients' web browsers remain the same during replay. In general, this might not be true if the user reacts differently to changes in the UI (e.g., some buttons might have changed during replay), but in all of the examples that we have considered, the user's interaction with the application is unchanged. In cases when the administrator knows about client-side changes that must be accounted for, Rail allows the administrator to supply a script to update the client inputs.

Third, Rail assumes that the mistakes leading to disclosures, either administrative or programming, are discovered before Rail's log rolls over.

Fourth, Rail deals only with data leaked through the web application. It cannot detect data revealed through other channels, such as an attacker directly querying the database or accessing the file system. Rail also cannot detect timing attacks, such as an attacker inferring a secret based on how long a response took.

Finally, Rail assumes that the software stack on the server is not compromised, which includes the operating system, system libraries, the web server, framework, and Rail itself. The adversary can, however, take advantage of vulnerabilities in the web application code.

5 System overview

At its core, Rail is a record and replay system. Rail views an application's execution as a stream of *actions*. Each action can read and write *objects*, such as database contents, session state, output, and non-deterministic inputs. This fine-grained view of an application's execution enables Rail to precisely track dependencies between actions and objects. This, in turn, allows Rail to replay a subset of actions when auditing, if it can determine that certain actions were not affected by a mistake. To maintain this dependency information, Rail records dependencies in a *log* during normal operation, and Rail's *replay controller* uses this log to decide what actions to replay for auditing.

Figures 4 and 5 summarize Rail's architecture and API, which we will outline in the rest of this section.

Action APIs. All application code in Rail is executed in the context of some *action*. Actions are the unit of dependency tracking and the unit of replay in Rail. Rail assumes that all application code runs in response to some event, such as an RPC request or a periodic timer event; there are no long-running threads. The web framework maintains a mapping between events and handlers for those events. For example, in Figure 1, the application registers two handlers: one for `pub_ans` subscription events, and one for `submit` RPC events. The handler for each event, stored by the web framework, is actually a named Rail object representing the code for that handler. The ex-

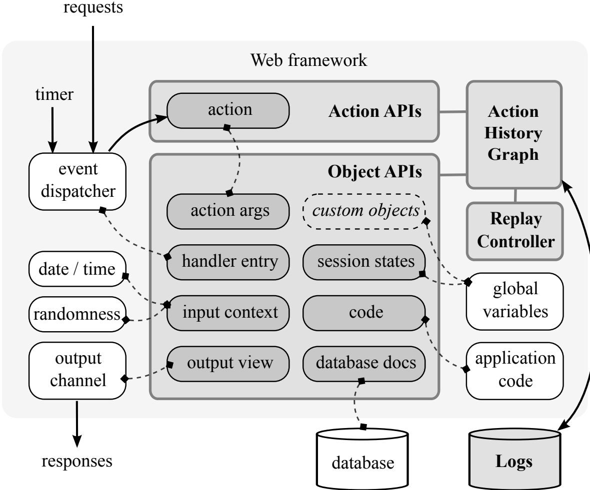


Figure 4: The architecture of RAIL. Strong shading indicates components introduced by RAIL. RAIL's object API constructs shadow objects for most of the shared state, inputs, and outputs in the web framework; these relationships are shown as dashed lines.

isting APIs provided by the web framework create these named object wrappers on the application's behalf; for example, both `App.publish` and `App.method` create such wrappers in Figure 1.

When the web framework receives an event, it retrieves the appropriate handler from its own tables, creates a new action to represent the execution of this event's handler, and invokes the handler in the context of this action, with the event as an argument. The last two steps are performed using the `doAction` API, as shown in Figure 5. In the example in Figure 1, the publish handler (lines 4–11) will run in a new action in response to each subscription request, and the `submit` method handler (lines 12–20) will run in a new action for every `submit` RPC request.

Each action has a *timestamp*, the time at which the action is triggered. Since the handler is a wrapper object, as described above, when the web framework invokes the handler, the handler first records a dependency from the handler object's name to the current action, and then runs the code wrapped by the object. This helps RAIL determine which actions need to re-execute when some code object changes.

Object APIs. Every shared object in RAIL, such as a database record, a function (code) object, or a session variable, is identified by a globally unique name. For each shared object, RAIL maintains two things: first, a set of dependencies between actions and objects, used to track down the set of actions that accessed an object during recording, and second, multiple versions of the object's state at different points in time, used during replay to implement rollback and to check for equivalence.

RAIL assumes that all application code uses object accessors to read and write shared objects, so that RAIL can track the input and output dependencies of actions, and can checkpoint the state of an object at different times. RAIL wraps existing framework objects using accessors, so that in most cases there is no need for the application developer to change the application code. For instance, on lines 6 and 13 of Figure 1, the application code uses the web framework's interface to access the user ID for the current session, which is session-level shared state.

RAIL provides an API for naming and accessing shared objects, which is used both by application code and by web framework code. The `findObject()` function returns a shared object given its unique name. Applications can perform two kinds of operations on a shared object: they can either read it, using `getValue()`, which registers a dependency *from* the object to the current action, or they can modify it, using an object-specific mutator, which registers a dependency *to* the object from the current action, and also records a checkpoint of the object's value.

This object API is used to handle dependencies for different kinds of objects, as we describe in more detail in §6.1. Some RAIL shared objects actually hold the state represented by the object. For example, this is the case for session state objects (accessed by the `getSessionUserId` method in Figure 1). In such situations, RAIL takes care of checkpointing, rollback, etc. In other cases, the RAIL shared object is just a placeholder, and the actual value is stored elsewhere. For example, this is the case for objects representing database state (where checkpointing and rollback takes place in the database, as opposed to in RAIL's log). This is also the case for code objects, since it is difficult to store a JavaScript closure in a log and restore it later on. Each object type defines its own mutators, using the `defineMutator` function; we will discuss these in more detail in §6.2.

Logs and dependency graph. RAIL's dependency graph is an *action history graph* [7] that connects action and object nodes. An edge from object o to action a means o is a's input ($o \rightarrow a$, or a reads o). Conversely, an edge from a to o means o is a's output ($a \rightarrow o$, or a writes o).

Since an object's state can change over time, dependencies in the action history graph refer to an object at a particular time. More precisely, $o \rightarrow a$ indicates that a depends on o's state right before time t_a, where t_a is a's timestamp. Here, RAIL assumes that actions are atomic, since all dependencies to and from an action effectively take place at a single instant in time. This is a reasonable assumption if the web framework provides serializability, which is true in the Meteor framework that our RAIL prototype is built on.

During an action's execution, RAIL connects edges from and to the current action's node as the action accesses objects. When the action completes, RAIL appends an entry to its persistent log, which contains the action's timestamp, its arguments (from the event), and the names

Return type	API	Description
	Public APIs for web framework and application developers	
–	doAction(*args*, *func*)	Allocate a new action and run *func* within its context.
action	getCurrentAction()	Return the current running action.
object	findObject(*id*)	Create or return the RAIL object identified by *id*.
<any>	*object*.getValue()	Accessor. Return the object's current state and update dependency.
function	defineMutator(*func*)	Return a mutator function based on *func*, which alters the binding object's state and updates dependency.
–	registerObjectType(*type*, *proto*)	Register a custom object *type* using template object *proto*.
function	registerCode(*id*, *func*)	Shortcut for creating a code object with the given *id*; returns a wrapper function that takes care of dependency tracking.
object	inputContext(*args*, ...)	Shortcut for creating an input context object identified by *args*.
	Private APIs for the replay controller	
number	*action*.timestamp	t_a, the timestamp when *action* starts. Also used to identify the action.
list<object>	*action*.reads	List of objects that the *action* depends on (inputs).
list<object>	*action*.writes	List of objects that depend on the *action* (outputs).
object	*action*.args	Return the argument object associated with the *action*.
–	replayAction(*action*)	Re-execute the given *action* based on its *args*.
string	*object*.type	The type name of the *object*.
number	*object*.time	The timestamp of the *object*'s current state during replay.
list<action>	*object*.actions	List of actions that read or write the *object*.
boolean	equiv(*object*, *ts*)	Check if the *object*'s current state is semantically equivalent to its state at time *ts* during original execution.
–	rollback(*object*, *ts*)	Revert *object*'s current state to its state at time *ts* during original run.

Figure 5: List of RAIL APIs.

of its input and output objects. The action history graph can be reconstructed from the log during replay. RAIL also logs every object mutation, so that during replay it can reconstruct the object's state at any instant. Objects that do not store actual state in the RAIL's shared object must maintain their own versioning outside of RAIL's log.

Replay controller. During auditing, the site's administrator initiates replay through the *replay controller*, by supplying either a code patch (e.g., fixing a software vulnerability), or a short JavaScript program that fixes the state of the system (e.g., correcting a mistake in an access control list). The administrator can also manipulate the action history graph and the versioned database state through JavaScript APIs, if necessary. The replay controller, in turn, reconstructs the action history graph from the log, and replays the relevant actions that were affected by the administrator's change, as we describe in §7.1. The replay controller computes the *view* of each session during replay, which represents the set of data objects sent to that client, and compares the views during replay with those during the original execution. Data objects that no longer show up in the view during replay are reported as inappropriate data disclosures.

6 Shared objects

To simplify dependency tracking and replay, RAIL defines a uniform API for managing different shared objects.

Object type	Naming convention
Action argument	args/<action id>
Code	code/<identifier>
Handler table entry	handler/<table>/<key>
Database document	db/<collection>/<doc id>
Session user ID	userid/<session id>
Session subscriptions	subs/<session id>
Session output view	view/<session id>
Input context	input/<action id>/<context id>

Figure 6: List of built-in object types.

6.1 Object types

RAIL identifies objects by globally unique names in the form of "`object_type/path_name`". There are several predefined object types in RAIL, as shown in Figure 6. These types represent most of the abstractions exposed by the web framework. When needed, the developer can also define their own object types using the `registerObjectType` API. In the rest of this subsection, we will describe what each object type represents.

Action argument. Every action depends on an argument object associated with it. If the action is triggered by a client request, for example, the argument contains the request message. Argument objects are immutable during replay, but the administrator can alter them before replay so as to force certain actions to be re-executed. For example, to cancel a request that creates a malicious account,

the administrator can change the corresponding action argument object to a null request.

Code object. RAIL must be able to determine which actions executed a given piece of code, so that if the code turns out to be buggy, RAIL can replay just the actions that may have been affected by that bug. To do this, RAIL uses a *code object* for every piece of application code, and records a dependency between an action and the code object when the action invokes the code.

RAIL creates code objects at function granularity, because it is easy to interpose on function invocation through a wrapper. The wrapper, created by the `registerCode` API function, records a dependency on the unique identifier of the function's code object, and then executes the function. This ensures that even if an action invokes many functions, the action history graph will contain dependencies to all functions invoked by that action.

RAIL automatically wraps global functions, and names the corresponding objects `code/`*`filename`*`/`*`funcname`*. For anonymous functions supplied as callbacks, the developer must assign a name to the anonymous callback in the function that accepts the callback argument. For example, in Figure 1, the `App.publish` function assigns the name `code/publish/pub_ans` to its anonymous callback, and the `App.method` function assigns the name `code/method/submit` to its anonymous callback.

During replay, RAIL's replay controller checks if any of the code objects have changed by comparing the textual representation of the new code object to the original textual representation of the code object as recorded in the log. If any of the code objects have been modified, the replay controller marks all of the actions that executed that code for replay. The textual representation of a function is insufficient to compare closures—e.g., references to variables in outer scopes are not well-defined in the textual representation. However, this is not a problem in JavaScript, because the only way to create an outer scope is to define another function, and if the outer scope of a function changes, the textual representation of that outer scope's function will be different, and will be flagged for replay by RAIL.

Handler table. In addition to tracking dependencies on functions, RAIL also needs to keep track of dependencies on the handler for a given type of event. For example, in Figure 1, the application developer may register a different, non-anonymous function as the handler for the `submit` RPC method. In this case, if the handler for the `submit` RPC method changes during replay, RAIL must detect this and replay all subsequent `submit` RPC invocations. To do this, the RAIL web framework creates a *handler table* object for every kind of handler registered in the web framework. For example, in Figure 1, `App.publish` records a dependency to the `handler/publish/pub_ans` object, and `App.method` records a dependency to the `handler/method/submit` object. The handler table object's value contains the function that will be invoked for that event (which, in practice, is likely to be a code object wrapper).

Database documents. RAIL assumes that the web application uses a key-value store as its persistent storage. Each data item, namely a *document*, has a unique identifier and other mutable fields. However, RAIL's approach is general enough so that it is also applicable to other storage models, including SQL databases and file systems. In particular, every database document is represented by an object named `db/`*`collection`*`/`*`docid`*. For efficiency, the RAIL web framework does not store the actual data in the RAIL database object; instead, the RAIL object is a placeholder for dependency tracking, and the actual data is stored, versioned, and rolled back in the database.

Output channel view. RAIL models a view of each session (i.e., the set of data items sent to that client) as a separate view object. View objects accumulate all data items disclosed through the corresponding output channel. By adding a data object, such as a database document, to the view, the application or framework code records that it sent the current state of the object through the output channel associated with the view. The RAIL web framework implements two types of view objects: a *session view* object, which represents all documents sent to a web browser, and an *email view* object, which represents all documents sent to a particular email address. Application developers can define new view objects for other types of output channels.

Other shared state. Accesses to in-memory global state, either application-level or session-level, should also go through the object API. Currently, RAIL defines two types of session state objects: *current user ID objects* and *subscription objects*. The current user ID object stores the logged-in user's identifier for each session. Subscription objects are necessary for interactive web applications that adopt a publish–subscribe pattern. They store a list of database queries that a session is interested in, so that whenever the results of any of these queries change, the web framework can notify the client about the updates.

Input context. Input context objects handle non-deterministic inputs requested by an action, such as current date and random numbers. They are important for stable re-execution, as we will discuss in §7.2.

6.2 Accessors and mutators

Every object has an accessor and a few mutators. Mutators vary with object types. For example, session user ID (`userid`) objects have two mutating methods:

`login(userid)` assigns the given user ID to the object's current state, and `logout()` resets its current state to `null`.

During normal execution, the accessor connects the object to the current action in the action history graph, and returns the current state of the object. Similarly, mutators connect the current action to the object, and change the current state of the object accordingly. In addition, mutators also log the mutating operation, so that by replaying the log during re-execution, RAIL can reconstruct checkpoints for all history states of the object.

During re-execution, accessors and mutators behave differently than during normal execution. If an object has been rolled back, the accessor returns the object's latest state; otherwise it returns the checkpoint state right before the current action. Mutators do not log changes during re-execution, but roll back the object before updating the object (see TRYROLLBACK in Figure 7). Since two executions are not identical, replay can introduce new dependencies that did not show up in the original execution. RAIL must keep updating the action history graph during replay to capture the new dependencies.

For performance reasons, accessors and mutators for database document objects are handled differently. RAIL employs a *time-travel database* [2] to keep every version that ever existed for each document in the database. Different versions of the same document are distinguished by two additional fields, `start_ts` and `end_ts`, which indicate the time interval within which the version is valid. Application code uses the web framework's database API to access the database as before. RAIL interposes on query processing and cursor accesses such that only the desired version is returned or updated. RAIL also performs dependency bookkeeping for the corresponding placeholder object of each affected document.

7 Replay

In order to determine what data was inappropriately disclosed, RAIL must re-compute the view objects for every session, and if it detects any session whose new view object is not a superset of the old view object, it reports the difference as a leak. Note that new data disclosed during replay does not result in a report.

RAIL recomputes the view objects by replaying previously recorded events and re-executing the corresponding actions. There are two challenges in doing so. First, for efficiency, RAIL should not re-execute every action; to this end, RAIL implements selective re-execution (§7.1). Second, for precision, RAIL should minimize divergence between replay and the original execution; to do this, RAIL uses context-based matching (§7.2).

7.1 Selective re-execution

Figure 7 shows the pseudo-code for RAIL's selective replay algorithm, inspired by Retro [7]. The algorithm relies

```
 1: procedure INITIALIZEREPLAY
 2:     objects ← LOADLOGS()
 3:     for all o ∈ objects do
 4:             ▷ Admin might change code or argument objects
 5:         if o.type ∈ { "code", "args" } then o.time ← 0
 6:         else  o.time ← ∞
 7:     return objects
 8: procedure NEXTACTION(objs)
 9:     acts ← {a | ∀o ∈ objs (a ∈ o.actions ∧ t_a > o.time)}
10:     if acts = ∅ then return nil
11:     return argmin_a{t_a | a ∈ acts}
12: procedure TRYROLLBACK(o, t)
13:     if o.time > t then
14:         ROLLBACK(o, t)
15:         o.time ← t
16: procedure MOVEFORWARD(o, t)
17:     if EQUIV(o, t + 1) then
18:         ROLLBACK(o, ∞)
19:         o.time ← ∞
20:     else  o.time ← t
21: procedure SELECTIVEREPLAY
22:     objects ← INITIALIZEREPLAY()
23:     a ← NEXTACTION(objects)
24:     while a ≠ nil  do
25:         C_in ← ∃o ∈ a.reads (o.time < ∞ ∧ ¬EQUIV(o, t_a))
26:         C_out ← ∃o ∈ a.writes (o.time < t_a)
27:             ▷ Replay if either inputs or outputs are changed
28:         if C_in ∨ C_out then
29:             for all o ∈ a.writes do TRYROLLBACK(o, t_a)
30:             REPLAYACTION(a)
31:             for all o ∈ a.writes do MOVEFORWARD(o, t_a)
32:         for all o ∈ a.reads do o.time ← max(o.time, t_a)
33:         a ← NEXTACTION(objects)
```

Figure 7: The selective replay algorithm. The algorithm uses private RAIL APIs listed in Figure 5.

on the *time* variable of each object, which indicates the timestamp of an object's current state and controls the progress of the re-execution.

Initially, every object is in its latest state ($time = \infty$), except for code and action argument objects, which the administrator could change to kick off replay.

In each round, RAIL picks the first action (action with the minimal timestamp) from a set of candidate actions to replay. Candidate actions are actions that read or wrote an object, and whose timestamps are bigger than the object's current *time*, meaning that they happen after the object's current state. Then, RAIL checks if the picked action needs re-execution. If any of its inputs have changed (C_{in} is true), RAIL must rerun it to generate new outputs; similarly, if any output has been rolled back to a state before the action took place (C_{out} is true), RAIL must rerun the action to reconstruct the output. Otherwise, RAIL can skip the action, and advance the timestamps for all of its inputs, so that the same action will not be selected again.

To replay an action, RAIL first rolls back all of the action's output objects recorded during the original execu-

tion to the state right before the action. This is important because the replayed action might not update the same objects as original. Then Rail reruns the action and updates the timestamps for the action's output objects. Note that during replay, mutators might roll back other output objects which were not captured in the original execution. As an optimization, if after replaying an action, an object's state is equivalent to its next state in the original execution, Rail will directly roll forward the object to its latest state to avoid considering actions that access the object in the future.

The selective re-execution algorithm is guaranteed to terminate because Rail always chooses the earliest available action. After each iteration, every relevant object will have a timestamp which is no smaller than the chosen action's. Therefore the timestamp of the picked action in each iteration monotonically increases.

Because of Rail's precise dependency tracking, the selective re-execution algorithm can minimize the number of actions replayed to just those that may have been affected by the mistake that triggered the audit. In our experience, selective re-execution replays only a small fraction of the total number of recorded actions.

One concern of selective re-execution is dealing with patches that significantly alter the control flow of an application. Rail works in this scenario because its unit of replay is an individual action (e.g., a client RPC request), and Rail's report is based on the set of objects that end up in a session's view. As long as the original and patched code add the same objects to the view, no disclosures will be reported regardless of code changes. In the case that the new code accesses many different shared objects, such as by issuing new database queries, Rail will replay more actions due to additional dependencies.

7.2 Context matching

Rail's goal is to precisely identify inappropriately disclosed data. Since Rail computes the set of disclosed data items as the difference between the original and the replayed view objects, Rail will report a data object as inappropriately disclosed if it fails to show up in the replayed view. This is desirable if the data item fails to show up in the replay view due to a fixed vulnerability. However, this is undesirable if it is a result of non-determinism, and some other choice of non-deterministic inputs could have led to the data *not* being flagged as disclosed.

This problem is made more complicated by the fact that *some* inputs to an action may have changed during replay. To minimize false reports, programmers must ensure that during replay, the behavior of non-deterministic code in the application remains as close as possible to that of the original execution, even in the face of input changes. We refer to this property as *application stability*. To help application writers to achieve this property, Rail provides helpful APIs. In some cases, using these APIs, it is easy for the application writers to achieve application stability, while in other cases it requires some thought. We provide a few examples to illustrate the issues in achieving application stability. Note that if an application does not achieve application stability, Rail will work correctly, but may generate false reports.

```
1      App.method('populate_admins', function() {
2  -     var admins = ['Alice', 'Mallory', 'Bob'];
3  +     var admins = ['Alice', 'Bob'];
4        for (var i = 0; i < admins.length; ++i) {
5          var pwd = Math.random();
6          /* BETTER: var pwd = Rail.inputContext(
7                'populate', admins[i]).random(); */
8          Users.insert({name: admins[i], passwd: pwd});
9        }
10     });
```

Figure 8: Example code demonstrating the necessity of using context identifiers to retrieve non-deterministic inputs.

For some application functions, it is relatively easy to make them stable. For example, in Figure 1, the `submit` RPC handler checks the current time (to see if the submission is late), and assigns a random ID to the submission. To ensure stability for this code, the programmer creates an *input context* object on line 15, which instructs Rail to reuse that same randomness during replay.

As a more complex case, consider the code fragment and patch shown in Figure 8. The code is intended to populate the database with a few predefined administrator accounts. Suppose that the site administrator later found out that Mallory was not supposed to have administrative privileges, and she wanted to see what information may have been disclosed as a result of this mistake. She does this by running Rail in auditing mode with Mallory removed from the list (see the change on lines 2–3 in Figure 8). Since Mallory was in the middle of the list, a simple heuristic that returns non-deterministic outputs in the same order as they were requested in the original run would reuse Mallory's password (the second invocation of `Math.random()`) for Bob during replay (since it is now the second invocation of `Math.random()`). Thus Bob's recorded login requests will fail during replay, causing many data items to be flagged as leaks. Prior systems such as Retro [7] and Warp [2] use this heuristic.

Rail tackles this problem by asking the programmer to assign stable *context identifiers* to non-deterministic inputs. During replay, Rail supplies non-deterministic values from the same context ID. Moreover, the current action's timestamp is also considered part of any context ID, so that all non-deterministic inputs are local to each action. Non-deterministic values with the same context ID are supposed to be semantically equivalent, therefore the programmer should make sure that the identifiers they choose are semantically stable. As an example, in the comments in lines 6–7 of Figure 8, we use the account

```
function pair_reviews(ids) {
  var seeds = [];
  for (var i = 0; i < ids.length; i++) {
    var ctx = Rail.inputContext('seed', ids[i]);
    seeds[i] = { sid: ids[i], rnd: ctx.random() };
  }
  var shuffle = _.sortBy(seeds, function (e) {
    return e.rnd;
  });
  for (var i = 0; i < ids.length; i++) {
    var reviewer = shuffle[i].sid;
    var reviewee = shuffle[(i+1) % ids.length].sid;
    var ctx = Rail.inputContext('pair', reviewer);
    Pairings.insert({ _id: ctx.random(),
        reviewer: reviewer, reviewee: reviewee });
  }
}
```

Figure 9: Illustration of a stable pairing algorithm.

name as part of the context identifier, which effectively ensures that the same account name will get the same password during replay, even if the list order has changed. In contrast, including the loop iterator `i` in the context ID is a bad idea, because it does not preserve semantics.

In some cases, making a function stable is an even more difficult problem. Consider the problem of pairing up students in a homework submission system for peer review. If one of the students is removed during replay, the set of pairwise assignments produced by most pairing algorithms would be quite different. To solve this problem, the programmer must devise a stable algorithm using context identifiers. Figure 9 illustrates such an algorithm that we designed for the homework submission application. The algorithm works by assigning every student a random pairing order, and then sorting the students by this pairing order. The pairing order is chosen through a context identifier tied to the user's username. This ensures that if students are added or removed, the overall sorted order is largely the same. Students are then paired up with other students next to them in this sort order. As a result, if a student is added or removed, this results in only a small number of changes to the overall pairings.

8 Implementation

We implemented a prototype of Rail on top of the Meteor web framework. Meteor has a clean interface for exchanging data between browser and server, which allows Rail to clearly identify data items. The core of the prototype is a standalone package that implements Rail's action APIs and object APIs.

The core package also maintains the action history graph and on-disk logs in two B-tree-like data structures—one stores actions (indexed by timestamps) and the other stores objects (indexed by object identifiers). Edges between actions and objects are stored twice (in both B-trees). During replay, Rail reconstructs the graph progressively without scanning the entire log. The prototype caches recently used B-tree blocks in memory and writes back dirty blocks in the background.

The prototype changes a few built-in packages in Meteor, so that accesses to standard Meteor abstractions are wrapped using Rail APIs. These abstractions include session user IDs, RPC dispatchers, session subscriptions, and MongoDB documents. Application developers can use standard interfaces to access these objects as before.

The prototype also includes a code rewriter, which automatically names and wraps top-level JavaScript functions using Rail's code objects when the application is loaded.

Our prototype consists of about 3,800 lines of JavaScript code, of which 2,987 lines are in the core package, 422 lines are for Meteor integration, and another 358 lines are for the code rewriter and command line tools.

9 Evaluation

We evaluate the Rail prototype with three real-world applications under synthetic attack workloads. Our evaluation aims to answer the following questions:

- What is the effort to port applications to Rail? (§9.1)
- What attack scenarios can Rail handle? (§9.2)
- How precise are Rail's data disclosure reports? (§9.3)
- What are Rail's performance and storage overheads during recording? (§9.4)
- How do the techniques described in §7 improve Rail's accuracy and performance for auditing? (§9.5)

9.1 Applications and developer effort

We ported three real-world web applications to Rail. Two of them are privacy-sensitive: one is *Submit*, a website that manages homework and grades, written by course staff from our department; the other is *EndoApp*, a medical survey application. Both applications run in production and have dozens to hundreds of users. We also ported *Telescope*, a widely used open-source social news application, to see how well Rail can support a full-fledged application with a relatively large code base.

Figure 10 summarizes the effort for porting these applications to Rail APIs. We had to modify fewer than 25 lines of code for each application. Most of the changes are related to non-deterministic inputs: programmers must provide context identifiers when generating date and random numbers in the application.

For Submit, modifications of two staff-only method handlers were necessary to ensure auditing correctness. First, the `getGrades` method summarizes grades of each assignment for all students, and returns a grid of grades directly to the client (not using the standard publish–subscribe mechanism). We rewrote the code using Rail's object API to explicitly add revealed data items to the current session's view object. Second, we modified the `pairing` function, as described in §7.2.

Appliation	Description	LoC (in JavaScript)		
		changed	server	client
Submit	homework grading	24	769	891
EndoApp	medical survey	2	599	900
Telescope	social news	20	1,169	1,781

Figure 10: Real-world web applications used in our evaluation, and the developer effort to port them to Rail APIs. We do not count HTML/CSS and third-party library code. Only server-side code is modified.

9.2 Attack case study

To evaluate whether Rail can identify disclosures after an attack, we chose the following common mistakes that can lead to data breaches in real-world settings.

Access control list error. In Submit, a course staff member erroneously grants "staff" privileges when creating a student account. The student logs in with this account and sees other students' homework solutions and grades. The staff later realizes the mistake, rectifies the initial request that created the account, and wants to know what unintended information has been revealed to the student. Rail identifies the leaks because during re-execution, the student's subscription request will be rejected by the server given the correct user privilege.

Stolen password. In EndoApp, a careless surgeon chooses a weak password, which is obtained by an outside attacker. Managing to stay concealed, the attacker creates another surgeon account, and logs in to the new account several times to retrieve sensitive patient profiles. After the administrator discovers the breach, presumably by looking for logins from unintended IP addresses, she cancels the suspicious login request to the careless surgeon's account, and wants to know what has been disclosed as a result of the suspicious login. Rail reports all breaches from both accounts, because without the initial login, all subscriptions and the account creation request would fail. All subsequent logins to the new account would also be denied since that bad account no longer exists.

Code bugs. This attack is based on a real bug in Telescope's commit history [4], in which the application performs permission checks according to a client-supplied current user ID, and publishes sensitive user emails for all accounts based on the flawed security check. An attacker can exploit the bug by executing JavaScript code with a chosen user ID from the browser console. After patching the code, the administrator wants to know if anyone exploited the bug, and whose emails were leaked. Rail detects the code change and reruns all subscription requests that depend on the code. The malicious request will be rejected during re-execution, while the legitimate ones (where the supplied user ID is the same as the session user ID) will return the same result as before. Therefore Rail can precisely identify leaked data from sessions that truly exploited the bug.

9.3 Auditing precision

To see if Rail can precisely report leaked data, we run the three attack workloads in parallel with other benign workloads in the background as noise. For each attack workload we consider two traces: a short trace in which background workloads stop soon after the attack, and a longer trace where benign accesses continue for several minutes. We compare the total number of data items accessed during the trace with the number reported by Rail. We manually inspect the report to count false reports (legitimate disclosures flagged by Rail as inappropriate) and missed ones (inappropriate disclosures according to our knowledge of the workload *not* reported by Rail). The result is shown in the last group of columns in Figure 11.

The number in the "accessed" column simulates what an access log based system, like Keypad, would report. As we can see, Rail precisely differentiates disclosures to the attacker from disclosures to legitimate users, resulting in fewer reports. Rail's report is stable: the duration of the trace and when the attack begins in the trace do not lead to more false reports.

After changing applications to use the Rail APIs, as described in §9.1, we do not observe any missed disclosures in our experiment. There is a single false report, however, for EndoApp workload. The reported database item is the malicious surgeon account added by the attacker, which *is* an expected disclosure for other legitimate surgeons logged in after the attack, because surgeons automatically subscribe to all user accounts upon login. But Rail flags it because the malicious surgeon does not appear in the re-execution anymore. We believe this false report is acceptable since it is related to the attack, and also helps the system administrator to identify the vulnerability.

9.4 Performance and overhead

We measure the performance of Rail using two machines running recent versions of Debian Linux. The server has an Intel Core i7 3.3 GHz processor and 24 GB of RAM; the client has eight 10-core Intel Xeon E7-8870 2.4 GHz processors with 256 GB of RAM. The client and the server are connected via a 1 Gbps network. To get a stable result, we pin the web server process to a single core of the server machine. The client machine is significantly more powerful to allow us to run enough browser instances to saturate the server. We use Splinter to drive PhantomJS browsers for all experiments.

Performance during normal execution. We compare the performance of Rail during normal execution to the performance of an unchanged version of Meteor, using Submit as the benchmark. In the "browse" workload, each

Attack workload	Trace	Number of requests			Running time (seconds)				Number of data items			
		total	attacker	replayed	original	replay	exec.	other	accessed	reported	false	missed
ACL error	Submit.short	2,972	45	16	161.0	1.6	0.9	0.7	1,142	193	0	0
	Submit.long	12,366	45	16	664.0	3.2	2.0	1.2	1,121	193	0	0
Stolen password	EndoApp.short	2,967	25	42	149.0	0.9	0.6	0.3	1,871	137	1	0
	EndoApp.long	8,597	25	270	640.0	10.0	3.1	6.9	3,521	197	1	0
Code bugs	Telescope.short	1,426	14	20	113.0	1.2	0.9	0.3	23	10	0	0
	Telescope.long	7,763	14	833	603.0	61.2	25.9	35.3	23	10	0	0

Figure 11: Replay performance and auditing precision under various workloads. "attacker" counts requests from the attacker's session. "original" is the duration for recording the trace. Replay time is broken down into two parts: "exec." shows the time for re-executing actions; "other" shows the time spent in other parts of the replay loop (all except line 30 of Figure 7). "reported" counts distinct data items flagged by Rail, out of all data items accessed by the trace.

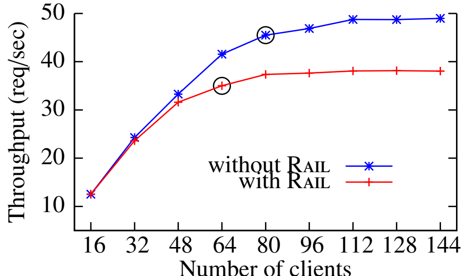


Figure 12: Server throughput when running Submit with the "browse" workload with an increasing number of concurrent clients. Circles mark the points where the server's average CPU usages exceeds 90%.

Workload	Throughput (reqs/sec)			Storage (KB/req)		
	w/o Rail	Rail	overhead	logs	DB	total
Browse	45.14	37.52	16.9%	0.34	0.12	0.46
Upload	45.50	37.36	17.9%	0.35	0.14	0.49

Figure 13: Performance and storage overhead during normal execution. Numbers are from a fully utilized server running Submit and serving 80 concurrent clients, which send requests as fast as possible. Two workloads with different write ratios are shown.

client repeatedly logs in using a random student account, browses the account's grades, and then logs out.

With a single client, the average latency for handling an individual request increases by 34% (from 15.0 to 20.1 msec). Profiling shows that executing wrappers, updating logs, and handling time-travel database queries contribute to the majority of the overhead.

To see how Rail performs under heavy workloads, we stress the server with an increasing number of clients, which send RPC requests as fast as possible. We measure the server's throughput and average CPU usage. As shown in Figure 12, the stock Meteor saturates at about 80 concurrent clients, while Rail saturates at 64 clients. For an under-loaded server (under 48 clients), Rail incurs less than 5% throughput overhead; for an over-loaded server (112 clients), the overhead is about 22%.

We also measure the throughput overhead for workloads with different write ratios. In the "upload" workload, there is a 20% probability that the user will submit a new answer to a homework after logging in (based on our historical logs), which leads to more write requests. Figure 13 shows the result: increasing the write ratio has a small impact for the overall performance, with the overhead going up from 16.9% to 17.9%.

Storage overhead. Figure 13 also shows the storage overhead. Rail's storage overhead consists of two parts: the compressed log, which contains the action history graph and the objects' mutation history, and the time-traveling database, which preserves all history versions of database records. The average overhead is 0.46 KB per request for the "browse" workload, and 0.49 KB per request for the "upload" workload. Note that login and logout also write to the database, updating login timestamps and tokens; this is why the numbers for the two workloads are close. With this overhead, a 500 GB disk can store 1 year worth of logs even for a fully utilized server. The time span is sufficient for most disclosure auditing tasks.

Replay performance. We measure Rail's replay performance using the traces shown in Figure 11. We consider two metrics: the number of replayed requests versus total number of requests, and the time to finish the replay, as shown in the first two groups of columns in Figure 11.

In Submit and EndoApp, Rail replays only a small fraction of all requests—just those related to the attack. For Submit, the number is even smaller than the total number of requests from the attacker's session, indicating that Rail's selective replay algorithm can effectively pick out just the relevant actions. In the Telescope workload, because the patched code is in the publish handler, Rail has to rerun subscription requests from all sessions, which causes each session's view object to be rolled back, and in turn triggers more replays. All of the re-executions are necessary to ensure that Rail captures all undesired leaks.

The "original" column shows the time to record each trace, which represents a typically loaded web server incurring about 20%–30% of CPU overhead. As we can see, Rail can replay lengthy traces in a small amount of time, and can report data leaks with high precision.

To understand what the performance bottleneck is during replay, we break down the replay time into two parts:

Technique	# of requests		# of leaked data	
	total	replayed	reported	false
Rail	103	5	2	0
w/o selective replay	103	93	2	0
w/o context matching	103	13	16	14

Figure 14: Impact of disabling each of Rail's techniques.

the time to re-execute actions (column "exec." in Figure 11), and the time spent in other parts of the replay loop ("other" in Figure 11), which comprises the overheads of the replay algorithm (including selecting actions, checking object equivalence, and updating object states). Both times increase as the number of replayed requests increases. For the "EndoApp.long" and "Telescope.long" workloads, the "other" time is higher than the "exec" time of re-executing actions. The "other" time, however, is time well spent: it is the overhead paid for avoiding re-execution of irrelevant actions.

9.5 Technique effectiveness

To demonstrate the value of selective re-execution and context identifiers, we use Submit with a setup of 30 students and one staff account. The staff member first creates a new account for a malicious student; then she initiates pairing, assigning each student (including the malicious one) two random reviewees using an algorithm similar to Figure 9. After five students log in and browse the reviews for their code, the staff runs Rail in replay mode after canceling the creation of the malicious account.

Figure 14 shows the result. Rail flags exactly two data items: one is the malicious user and the other is the pairing record for the user. Out of 103 requests, Rail replayed only five, which includes the pairing request and four requests from students paired with the malicious account.

Without selective re-execution, Rail reruns 93 requests in total, including all requests that follow the account creation. Disabling context matching introduces 14 false reports: as pairings change, most students see homework answers from different peers during replay; the IDs of pairing records will also be different, constituting the rest of the false reports. This demonstrates the importance of Rail's selective re-execution and context matching.

10 Discussion

This section discusses our experience with Rail.

10.1 Supporting Rail in other frameworks

Our Rail prototype demonstrates that by utilizing the rich semantics available in the web framework, one can achieve fine-grained information tracking at low cost. Although our prototype is based on a specific framework (Meteor), the design of Rail's core API is framework-independent. Rail's techniques can be applied to other web frameworks as long as they meet a few assumptions.

First, the framework should force developers to use the framework's abstractions and APIs to access web objects, such as requests, responses, sessions, databases, and files. Bypassing these interfaces should be considered rare or prohibited entirely. This helps Rail interpose on accesses to these objects at a level where useful semantics are preserved. Adopting Rail to another framework involves wrapping the framework's object APIs with Rail APIs.

Second, the framework should provide a mechanism that separates data items from their web representation. Meteor attains the separation by sending data items directly over the wire, and constructing web pages purely on the client side. Other commonly used frameworks, such as Ruby and Django, do not share this paradigm. However, they do adopt the model-view-controller (MVC) pattern using server-side template rendering systems that clearly separate data and views. The major difference in porting Rail to these frameworks lies in how to track responses: one could wrap the template rendering system (as opposed to the publish system in Meteor) with Rail's output view object API to capture revealed data.

Third, the framework should maintain as little global state as possible. To correctly support selective re-execution, Rail must interpose on accesses to all global objects in order to track dependencies and make continuous checkpoints. Excessive use of global state can introduce false dependencies among requests and increase space overhead. Fortunately, most web frameworks do not maintain global state other than the persistent storage (e.g., the database) and a simple session store in their core packages. External packages, however, might keep their own shared state. As an example, Meteor's account package does not reuse Meteor's session store, but keeps per-session authentication tokens on its own. When porting Rail to a new framework, one must also examine external packages to ensure that all package-defined global objects are properly wrapped.

Finally, Rail's current design has a simplified API that assumes action serializability. We believe this captures an important class of real-world web applications: for instance, Node.js applications fall into this sequential execution model. Nevertheless, Rail's API could be extended to support concurrent action execution. This would require finer-grained dependency tracking and replay at a lower level. For instance, one could treat each access to a shared object (e.g., database query) as atomic, and record dependencies between such operations. During replay, each action might be interleaved with the replay of other actions. This is similar to how multi-threaded record-replay systems work, and to how Retro [7] dealt with record-replay of concurrent processes.

10.2 Porting applications

Porting an application to Rail is easy, because the framework wrappers do most of the work, such as wrapping and trapping accesses to global objects. In rare cases, if an application defines its own class of global objects, the programmer must wrap accesses to these objects using the Rail API.

For most applications, no matter how large the code size is, the majority of changes will be for handling non-deterministic input. Since application stability must exploit high-level knowledge unavailable in the code, it cannot be implemented without help from developers. For example, no one knows better than the developer what the context identifier should be for a non-deterministic value. Identifying non-determinism in the code could be a potential challenge when porting applications.

Fortunately, there are only a handful of sources of non-deterministic input that have to be handled—for most cases they are date, time, and random numbers. These values usually come from the language's library calls, such as `now()` and `random()`. Simply hiding these library interfaces from developers could help them identify sources of non-determinism and force them to use Rail's wrappers.

Simple program analysis can also help identify these sources of non-determinism, and can be used to suggest context identifiers, as we will discuss next.

10.3 Choosing context identifiers

The goal of context IDs is to preserve application stability. As a general guideline, the context ID usually contains the primary key of the data item tied to the non-deterministic value, plus an optional string describing the purpose of the value. In this subsection, we illustrate this rule with examples we encountered in benchmark applications.

The most common use of context IDs is to generate random identifiers for new data items. For example, when generating a document ID for a new homework submission in Figure 1, the context ID should be the pair (*homework_ID*, *student_ID*), which uniquely identifies a homework submission. Similarly, when adding a comment in Telescope, the context ID should contain the topic ID and the user ID. If multiple random values are requested using the same primary key within a single action, one can add descriptive strings to distinguish different invocations, like on lines 4 and 13 of Figure 8. In practice, this process could be automated by a simple analysis of the database schema.

Another common use is to generate dates and timestamps. For instance, when a student adds a homework submission, the application needs to check the current date against the homework's deadline. Since the current date is not tied to any data item, we simply supply a constant string "`checkdeadline`" as the context identifier. The descriptive string helps distinguish this date query from others in the same action, if any. Often, the calling function's name and signature can be used as the descriptive string when requesting the current date.

Context IDs also play an important role in preserving cryptographic randomness. For example, Meteor's account package uses the SRP protocol to authenticate user logins. Internally, SRP generates random values, and if they are not preserved, login will not replay correctly. In this case, we use the encrypted password as the context ID when generating the random SRP verifier, so that the same login password yields the same verifier during replay.

10.4 Misuse of Rail APIs

Inadvertent misuses of Rail APIs might affect Rail's accuracy. Rail requires the developer to use the framework's standard interface to access global objects, such as the database. If the developer forgets to wrap application-defined global state with Rail APIs, Rail will miss the dependency, omitting relevant actions from selective replay. This will leave the replayed application in an inconsistent state, and likely lead to false negatives.

If the developer does not use Rail's wrappers to retrieve non-deterministic values, or inappropriately chooses context IDs, Rail will produce a different value during replay. Consequently, requests that depend on these non-deterministic inputs (such as logins) will behave differently. The divergence will cause more actions to be replayed, and introduce false positives and false negatives.

Note that the worst outcome of API misuse is unnecessary re-execution and inaccurate reports. The entire replay process is guaranteed to terminate.

11 Conclusion

Rail is the first system for precisely auditing past data disclosures in web applications. Based on rollback and replay, Rail introduces an explicit API that application developers must use to identify data items, track dependencies, and match up states. The API helps Rail minimize state divergence and unnecessary re-execution, providing fast and precise auditing. Measurements with a Rail prototype show that Rail can precisely distinguish legitimate data disclosures from illegal ones caused by human mistakes. Rail requires only minor changes to web applications and incurs a moderate performance overhead. Rail's source code is publicly available at `https://github.com/haogang/rail`.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Landon Cox, for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 1–14, San Antonio, TX, June 2003.

[2] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 101–114, Cascais, Portugal, Oct. 2011.

[3] R. Chandra, T. Kim, and N. Zeldovich. Asynchronous intrusion recovery for interconnected web services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 213–227, Farmington, PA, Nov. 2013.

[4] M. DeBergalis. Use this.userId() in publish rather than an arg, Sept. 2012. https://github.com/TelescopeJS/Telescope/pull/6.

[5] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[6] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proceedings of the ACM EuroSys Conference*, pages 1–16, Salzburg, Austria, Apr. 2011.

[7] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 89–104, Vancouver, Canada, Oct. 2010.

[8] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 193–206, Hollywood, CA, Oct. 2012.

[9] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, Feb. 2005.

[10] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: Secure offline data access using commodity trusted hardware. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[11] Meteor Development Group. Meteor: A better way to build apps. https://www.meteor.com/.

[12] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 157–172, Seattle, WA, Apr. 2014.

[13] M. Rile, B. Elgin, D. Lawrence, and C. Matlack. Missed alarms and 40 million stolen credit card numbers: How Target blew it. Bloomberg Businessweek, Mar. 2013. http://www.businessweek.com/articles/2014-03-13/target-missed-alarms-in-epic-hack-of-credit-card-data.

[14] X. Wang, N. Zeldovich, and M. F. Kaashoek. Retroactive auditing. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011.

[15] A. G. Wylie. University of Maryland: Data breach, Mar. 2014. http://www.umd.edu/datasecurity/.

[16] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, Apr. 2007.

Project Adam: Building an Efficient and Scalable Deep Learning Training System

Trishul Chilimbi Yutaka Suzue Johnson Apacible Karthik Kalyanaraman

Microsoft Research

ABSTRACT

Large deep neural network models have recently demonstrated state-of-the-art accuracy on hard visual recognition tasks. Unfortunately such models are extremely time consuming to train and require large amount of compute cycles. We describe the design and implementation of a distributed system called Adam comprised of commodity server machines to train such models that exhibits world-class performance, scaling and task accuracy on visual recognition tasks. Adam achieves high efficiency and scalability through whole system co-design that optimizes and balances workload computation and communication. We exploit asynchrony throughout the system to improve performance and show that it additionally improves the accuracy of trained models. Adam is significantly more efficient and scalable than was previously thought possible and used 30x fewer machines to train a large 2 billion connection model to 2x higher accuracy in comparable time on the ImageNet 22,000 category image classification task than the system that previously held the record for this benchmark. We also show that task accuracy improves with larger models. Our results provide compelling evidence that a distributed systems-driven approach to deep learning using current training algorithms is worth pursuing.

1. INTRODUCTION

Traditional statistical machine learning operates with a table of data and a prediction goal. The rows of the table correspond to independent observations and the columns correspond to hand crafted features of the underlying data set. Then a variety of machine learning algorithms can be applied to learn a model that maps each data row to a prediction. More importantly, the trained model will also make good predictions for unseen test data that is drawn from a similar distribution as the training data. Figure 1 illustrates this process.

This approach works well for many problems such as recommendation systems where a human domain expert can easily construct a good set of features. Unfortunately it fails for hard AI tasks such as speech recognition or visual object classification where it is extremely hard to construct appropriate features over the input data. Deep learning attempts to address this shortcoming by additionally learning hierarchical

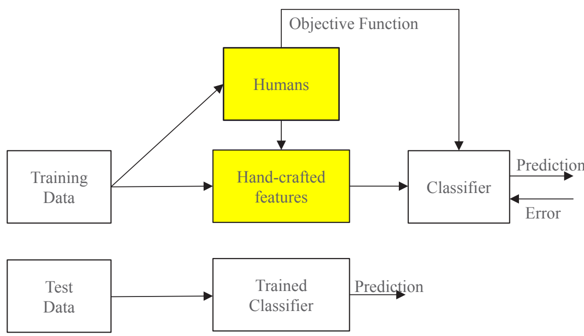


Figure 1. Machine Learning.

features from the raw input data and then using these features to make predictions as illustrated in Figure 2 [1]. While there are a variety of deep models we focus on deep neural networks (DNNs) in this paper.

Deep learning has recently enjoyed success on speech recognition and visual object recognition tasks primarily because of advances in computing capability for training these models [14, 17, 18]. This is because it is much harder to learn hierarchical features than optimize models for prediction and consequently this process requires significantly more training data and computing power to be successful. While there have been some advances in training deep learning systems, the core algorithms and models are mostly unchanged from the eighties and nineties [2, 9, 11, 19, 25].

Complex tasks require deep models with a large number of parameters that have to be trained. Such large models require significant amount of data for successful training to prevent over-fitting on the

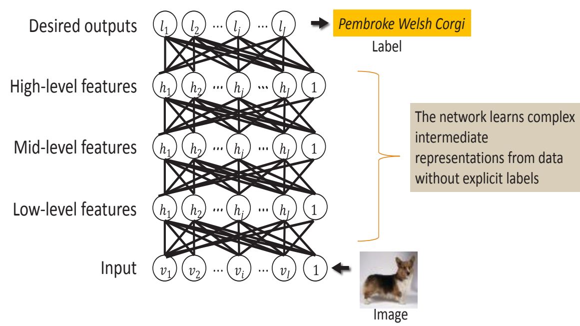


Figure 2. Deep networks learn complex representations.

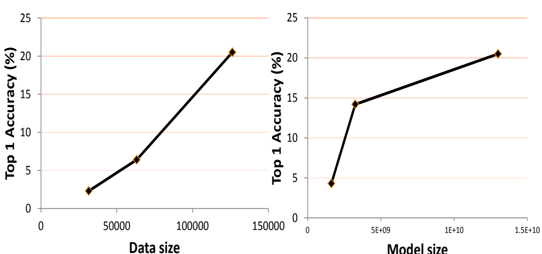


Figure 3. Accuracy improvement with larger models and more data.

training data which leads to poor generalization performance on unseen test data. Figure 3 illustrates the impact of larger DNNs and more training data on the accuracy of a visual image recognition task. Unfortunately, increasing model size and training data, which is necessary for good prediction accuracy on complex tasks, requires significant amount of computing cycles proportional to the product of model size and training data volume as illustrated in Figure 4.

Due to the computational requirements of deep learning almost all deep models are trained on GPUs [5, 17, 27]. While this works well when the model fits within 2-4 GPU cards attached to a single server, it limits the size of models that can be trained. To address this, researchers recently built a large-scale distributed system comprised of commodity servers to train extremely large models to world record accuracy on a hard visual object recognition task—classifying images into one of 22 thousand distinct categories using only raw pixel information [7, 18]. Unfortunately their system scales poorly and is not a viable cost-effective option for training large DNNs [7].

This paper addresses the problem by describing the design and implementation of a scalable distributed deep learning training system called Adam comprised of commodity servers. The main contributions include:

- *Optimizing and balancing both computation and communication for this application through whole system co-design*. We partition large models across machines so as to minimize memory bandwidth and cross-machine communication requirements. We restructure the computation across machines to reduce communication requirements.
- *Achieving high performance and scalability by exploiting the ability of machine learning training to tolerate inconsistencies well*. We use a variety of techniques including multi-threaded model parameter updates without locks, asynchronous batched parameter updates that take advantage of weight updates being associative and commutative, and permit computation over stale parameter values. Surprisingly, it appears that asynchronous training also improves model accuracy.
- *Demonstrating that system efficiency, scaling, and asynchrony all contribute to improvements in trained model accuracy*. Adam uses 30x fewer machines to train a large 2 billion connection model to 2x higher accuracy in comparable time on the ImageNet 22,000 category image classification task than the system that previously held the record for this benchmark. We also show that task accuracy improves with model size and Adam's efficiency enables training larger models with the same amount of resources.

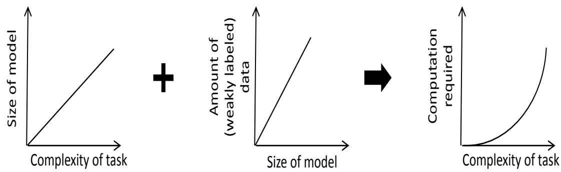


Figure 4. Deep Learning Computational Requirements.

Our results suggest an opportunity for a distributed-systems driven approach to large-scale deep learning where prediction accuracy is increased by training larger models on vast amounts of data using efficient and scalable compute clusters rather than relying solely on algorithmic breakthroughs from the machine learning community.

The rest of the paper is organized as follows. Section 2 covers background material on training deep neural networks for vision tasks and provides a brief overview of large-scale distributed training. Section 3 describes the Adam design and implementation focusing on the computation and communication optimizations, and use of asynchrony, that improve system efficiency and scaling. Section 4 evaluates the efficiency and scalability of Adam as well as the accuracy of the models that it trains. Finally, Section 5 covers related work.

2. BACKGROUND

2.1 Deep Neural Networks for Vision

Artificial neural networks consist of large numbers of homogeneous computing units called neurons with multiple inputs and a single output. These are typically connected in a layer-wise manner with the output of neurons in layer *l-1* connected to all neurons in layer *l* as in Figure 2. Deep neural networks have multiple layers that enable hierarchical feature learning.

The output of a neuron i in layer l, called the activation, is computed as a function of its inputs as follows:

$$a_i(l) = F((\Sigma_{j=1..k}\, w_{ij}(l\text{-}1,l)*a_j(l\text{-}1)) + b_i)$$

where w_{ij} is the weight associated with the connection between neurons i and j and b_i is a bias term associated with neuron i. The weights and bias terms constitute the parameters of the network that must be learned to accomplish the specified task. The activation function, F, associated with all neurons in the network is a pre-defined non-linear function, typically sigmoid or hyperbolic tangent.

Convolutional neural networks are a class of neural networks that are biologically inspired by early work on the visual cortex [15, 19]. Neurons in a layer are only connected to spatially local neurons in the next layer modeling local visual receptive fields. In addition, these connections share weights which allows for feature detection regardless of position in the visual field. The weight sharing also reduces the number of free parameters that must be learned and consequently these models are easier to train compared to similar size networks where neurons in a layer are fully connected to all neuron in the next layer. A convolutional layer is often followed by a max-pooling layer that performs a type of nonlinear down-sampling by outputting the maximum value from non-overlapping sub-regions. This provides the network with robustness to small translations in the input as the max-pooling layer will produce the same value.

The last layer of a neural network that performs multiclass classification often implements the softmax function. This function transforms an n-dimensional vector of arbitrary real values to an n-dimensional vector of values in the range between zero and one such that these component values sum to one.

We focus on visual tasks because these likely require the largest scale neural networks given that roughly one third of the human cortex is devoted to vision. Recent work has demonstrated that deep neural networks comprised of 5 convolutional layers for learning visual features followed by 3 fully connected layers for combining these learned features to make a classification decision achieves state-of-the-art performance on visual object recognition tasks [17, 27].

2.2 Neural Network Training

Neural networks are typically trained by back-propagation using gradient descent. Stochastic gradient descent is a variant that is often used for scalable training as it requires less cross-machine communication [2]. In stochastic gradient descent the training inputs are processed in a random order. The inputs are processed one at a time with the following steps performed for each input to update the model weights.

Feed-forward evaluation:

The output of each neuron i in a layer l, called its activation, a, is computed as a function of its k inputs from neurons in the preceding layer $l\text{-}1$ (or input data for the first layer). If $w_{ij}(l\text{-}1,l)$ is the weight associated with a connection between neuron j in layer $l\text{-}1$ and neuron i in layer l:

$$a_i(l) = F((\Sigma_{j=1..k}\, w_{ij}(l\text{-}1,l)*a_j(l\text{-}1)) + b_i)$$

where b is a bias term for the neuron.

Back-propagation:

Error terms, δ, are computed for each neuron, i, in the output layer, l_n, first as follows:

$$\delta_i(l_n) = (t_i(l_n) - a_i(l_n))*F'(a_i(l_n))$$

where $t(x)$ is the true value of the output and $F'(x)$ is the derivative of $F(x)$.

These error terms are then back-propagated for each neuron i in layer l connected to m neurons in layer $l+1$ as follows:

$$\delta_i(l) = (\Sigma_{j=1..m}\, \delta_j(l+1)*w_{ji}(l,l+1))*F'(a_i(l))$$

Weight updates:

These error terms are used to update the weights (and biases similarly) as follows:

$$\Delta w_{ij}(l\text{-}1,l) = \alpha*\delta_i(l)*a_j(l\text{-}1) \text{ for } j = 1 .. k$$

where α is the learning rate parameter. This process is repeated for each input until the entire training dataset has been processed, which constitutes a training epoch. At the end of a training epoch, the model prediction error is computed on a held out validation set. Typically, training continues for multiple epochs, reprocessing the training data set each time, until the validation set error converges to a desired (low) value.

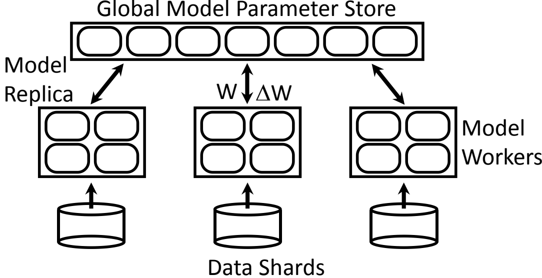


Figure 5. Distributed Training System Architecture.

The trained model is then evaluated on (unseen) test data.

2.3 Distributed Deep Learning Training

Recently, Dean et al. described a large-scale distributed system comprised of tens of thousands of CPU cores for training large deep neural networks [7]. The system architecture they used (shown in Figure 5) is based on the Multi-Spert system and exploits both model and data parallelism [9]. Large models are partitioned across multiple model worker machines enabling the model computation to proceed in parallel. Large models require significant amounts of data for training so the systems allows multiple replicas of the same model to be trained in parallel on different partitions of the training data set. All the model replicas share a common set of parameters that is stored on a global parameter server. For speed of operation each model replica operates in parallel and asynchronously publishes model weight updates to and receives updated parameter weights from the parameter server. While these asynchronous updates result in inconsistencies in the shared model parameters, neural networks are a resilient learning architecture and they demonstrated successful training of large models to world-record accuracy on a visual object recognition task [18].

3. ADAM SYSTEM ARCHITECTURE

Our high-level system architecture is also based on the Multi-Spert system and consists of data serving machines that provide training input to model training machines organized as multiple replicas that asynchronously update a shared model via a global parameter server. While describing the design and implementation of Adam we focus on the computation and communication optimizations that improve system efficiency and scaling. These optimizations were motivated by our past experience building large-scale distributed systems and by profiling and iteratively improving the Adam system. In addition, the system is built from the ground up to support asynchronous training.

While we focus on vision tasks in this paper, the Adam system is general-purpose as stochastic gradient descent is a generic training algorithm that can train any DNN via back-propagation. In addition, Adam supports training any combination of stacked convolutional and fully-connected network layers and can be used to train models on tasks such as speech recognition and text processing.

3.1 Fast Data Serving

Training large DNNs requires vast quantities of training data (10-100 TBs). Even with large quantities of training data these DNNs require data transformations to avoid over-fitting when iterating through the data set multiple times. We configure a small set of machines as data serving machines to offload the computational requirements of these transformations from the model training machines and ensure high throughput data delivery.

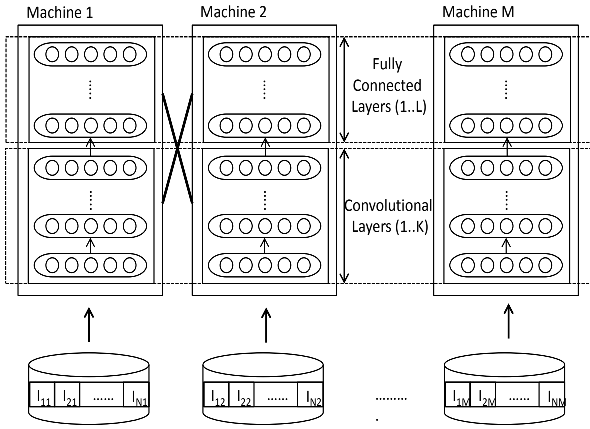


Figure 6. Model partitioning across training machines.

For vision tasks, the transformations include image translations, reflections, and rotations. The training data set is augmented by randomly applying a different transformation to each image so that each training epoch effectively processes a different variant of the same image. This is done in advance since some of the image transformations are compute intensive and we want to immediately stream the transformed images to the model training machines when requested.

The data servers pre-cache images utilizing nearly the entire system memory as an image cache to speed image serving. They use asynchronous IO to process incoming requests. The model training machines request images in advance in batches using a background thread so that the main training threads always have the required image data in memory.

3.2 Model Training

Models for vision tasks typically contain a number of convolutional layers followed by a few fully connected layers [17, 27]. We partition our models vertically across the model worker machines as shown in Figure 6 as this minimizes the amount of cross-machine communication that is required for the convolution layers.

3.2.1 Multi-Threaded Training

Model training on a machine is multi-threaded with different images assigned to threads that share the model weights. Each thread allocates a training context for feed-forward evaluation and back propagation. This

training context stores the activations and weight update values computed during back-propagation for each layer. The context is pre-allocated to avoid heap locks while training. Both the context and per-thread scratch buffer for intermediate results use NUMA-aware allocations to reduce cross-memory bus traffic as these structures are frequently accessed.

3.2.2 Fast Weight Updates

To further accelerate training we access and update the shared model weights locally without using locks. Each thread computes weight updates and updates the shared model weights. This introduces some races as well as potentially modifying weights based on stale weight values that were used to compute the weight updates but have since been changed by other threads. We are still able to train models to convergence despite this since the weight updates are associative and commutative and because neural networks are resilient and can overcome the small amount of noise that this introduces. Updating weights without locking is similar to the Hogwild system except that we rely on weight updates being associative and commutative instead of requiring that the models be sparse to minimize conflicts [23]. This optimization is important for achieving good scaling when using multiple threads on a single machine.

3.2.3 Reducing Memory Copies

During model training data values need to be communicated across neuron layers. Since the model is partitioned across multiple machines some of this communication is non local. We use a uniform optimized interface to accelerate this communication. Rather than copy data values we pass a pointer to the relevant block of neurons whose outputs need communication avoiding expensive memory copies. For non-local communication, we built our own network library on top of the Windows socket API with IO completion ports. This library is compatible with our data transfer mechanism and accepts a pointer to a block of neurons whose output values need to be communicated across the network. We exploit knowledge about the static model partitioning across machines to optimize communication and use reference counting to ensure safety in the presence of asynchronous network IO. These optimizations reduce the memory bandwidth and CPU requirements for model training and are important for achieving good performance when a model is partitioned across machines.

3.2.4 Memory System Optimizations

We partition models across multiple machines such that the working sets for the model layers fit in the L3 cache. The L3 cache has higher bandwidth than main memory and allows us to maximize usage of the floating point units on the machine that would otherwise be limited by memory bandwidth.

We also optimize our computation for cache locality. The forward evaluation and back-propagation computation have competing locality requirements in terms of preferring a row major or column major layout for the layer weight matrix. To address this we created two custom hand-tuned assembly kernels that appropriately pack and block the data such that the vector units are fully utilized for the matrix multiply operations. These optimizations enable maximal utilization of the floating point units on a machine.

3.2.5 Mitigating the Impact of Slow Machines

In any large computing cluster there will always be a variance in speed between machines even when all share the same hardware configuration. While we have designed the model training to be mostly asynchronous to mitigate this, there are two places where this speed variance has an impact. First, since the model is partitioned across multiple machines the speed of processing an image is limited by slow machines. To avoid stalling threads on faster machines that are waiting for data values to arrive from slower machines, we allow threads to process multiple images in parallel. We use a dataflow framework to trigger progress on individual images based on arrival of data from remote machines. The second place where this speed variance manifests is at the end of an epoch. This is because we need to wait for all training images to be processed to compute the model prediction error on the validation data set and determine whether an additional training epoch is necessary. To address this, we implemented the simple solution of ending an epoch whenever a specified fraction of the images are completely processed. We ensure that the same set of images are not skipped each epoch by randomizing the image processing order for each epoch. We have empirically determined that waiting for 75% of the model replicas to complete processing all their images before declaring the training epoch complete can speed training by up to 20% with no impact on the trained model's prediction accuracy. An alternative solution that we did not implement is to have the faster machines steal work from the slower ones. However, since our current approach does not affect model accuracy this is unlikely to outperform it.

3.2.6 Parameter Server Communication

We have implemented two different communication protocols for updating parameter weights. The first version locally computes and accumulates the weight updates in a buffer that is periodically sent to the parameter server machines when k (which is typically in the hundreds) images have been processed. The parameter server machines then directly apply these

accumulated updates to the stored weights. This works well for the convolutional layers since the volume of weights is low due to weight sharing. For the fully connected layers that have many more weights we use a different protocol to minimize communication traffic between the model training and parameter server machines. Rather than directly send the weight updates we send the activation and error gradient vectors to the parameter server machines where the matrix multiply can be performed locally to compute and apply the weight updates. This significantly reduces the communication traffic volume from $M*N$ to $k*(M+N)$ and greatly improves system scalability. In addition, it has an additional beneficial aspect as it offloads computation from the model training machines where the CPU is heavily utilized to the parameter server machines where the CPU is underutilized resulting in a better balanced system.

3.3 Global Parameter Server

The parameter server is in constant communication with the model training machines receiving updates to model parameters and sending the current weight values. The rate of updates is far too high for the parameter server to be modeled as a conventional distributed key value store. The architecture of a parameter server node is shown in Figure 7.

3.3.1 *Throughput Optimizations*

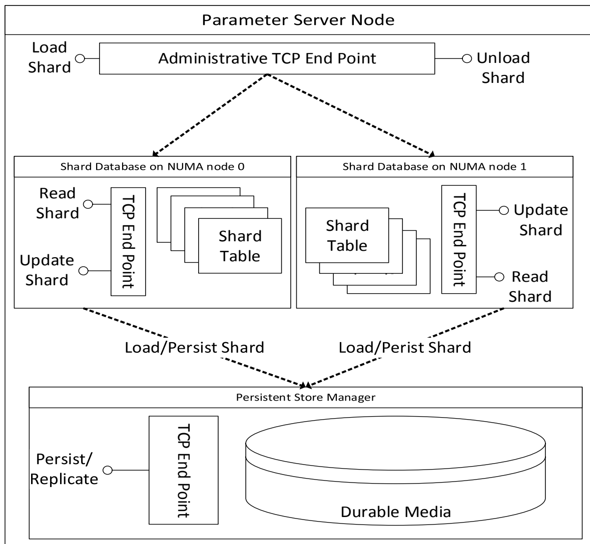


Figure 7. Parameter Server Node Architecture.

The model parameters are divided into 1 MB sized shards, which represents a contiguous partition of the parameter space, and these shards are hashed into storage buckets that are distributed equally among the parameter server machines. This partitioning improves the spatial locality of update processing while the distribution helps with load balancing. Further, we opportunistically batch updates. This improves temporal locality and relieves pressure on the L3 cache by applying all updates in a batch to a block of parameters before moving to next block in the shard. The parameter servers use SSE/AVX instructions for applying the update and all processing is NUMA aware. Shards are allocated on a specific NUMA node and all update processing for the shard is localized to that NUMA node by assigning tasks to threads bound to the processors for the NUMA node by setting the appropriate processor masks. We use lock free data structures for queues and hash tables in high traffic execution paths to speed up network, update, and disk IO processing. In addition, we implement lock free memory allocation where buffers are allocated from pools of specified size that vary in powers of 2 from 4KB all the way to 32MB. Small object allocations are satisfied by our global lock free pool for the object. All of these optimizations are critical to achieving good system scalability and were arrived at through iterative system refinement to eliminate scalability bottlenecks.

3.3.2 *Delayed Persistence*

We decouple durability from the update processing path to allow for high throughput serving to training nodes. Parameter storage is modelled as a write back cache, with dirty chunks flushed asynchronously in the background. The window of potential data loss is a function of the IO throughput supported by the storage layer. This is tolerable due to the resilient nature of the underlying system as DNN models are capable of learning even in the presence of small amounts of lost updates. Further, these updates can be effectively recovered if needed by retraining the model on the appropriate input data. This delayed persistence allows for compressed writes to durable storage as many updates can be folded into a single parameter update, due to the additive nature of updates, between rounds of flushes. This allows update cycles to catch up to the current state of the parameter shard despite update cycles being slower.

3.3.3 *Fault Tolerant Operation*

There are three copies of each parameter shard in the system and these are stored on different parameter servers. The shard version that is designated as the primary is actively served while the two other copies are designated as secondary for fault tolerance. The parameter servers are controlled by a set of parameter server (PS) controller machines that form a Paxos cluster. The controller maintains in its replicated state the configuration of parameter server cluster that contains the mapping of shards and roles to parameter servers. The clients (model training machines) contact the controller to determine request routing for parameter shards. The PS controller hands out bucket

assignments (primary role via a lease, secondary roles with primary lease information) to parameter servers and persists the lease information in its replicated state. The controller also receives heart beats from parameter server machines and relocates buckets from failed machines evenly to other active machines. This includes assigning new leases for buckets where the failed machine was the primary.

The parameter server machine that is the primary for a bucket accepts requests for parameter updates for all chunks in that bucket. The primary machine replicates changes to shards within a bucket to all secondary machines via a 2 phase commit protocol. Each secondary checks the lease information of the bucket for a replicated request initiated by primary before committing. Each parameter server machine sends heart beats to the appropriate secondary machines for all buckets for which it has been designated as primary. Parameter servers that are secondary for a bucket initiate a role change proposal to be a primary along with previous primary lease information to the controller in the event of prolonged absence of heart beats from the current primary. The controller will elect one of the secondary machines to be the new primary, assigns a new lease for the bucket and propagates this information to all parameter server nodes involved for the bucket. Within a parameter server node, the on disk storage for a bucket is modelled as a log structured block store to optimize disk bandwidth for the write heavy work load.

We have used Adam extensively over the past two years to run several training experiments. Machines did fail during these runs and all of these fault tolerance mechanisms were exercised at some point.

3.3.4 Communication Isolation

Parameter server machines have two 10Gb NICs. Since parameter update processing from a client (training) perspective is decoupled from persistence, the 2 paths are isolated into their own NICs to maximize network bandwidth and minimize interference as shown in Figure 7. In addition, we isolate administrative traffic from the controller to the 1Gb NIC.

4. EVALUATION

4.1 Visual Object Recognition Tasks

We evaluate Adam using two popular benchmarks for image recognition tasks. MNIST is a digit classification task where the input data is composed of 28x28 images of the 10 handwritten digits [20]. This is a very small benchmark with 60,000 training images and 10,000 test images that we use to characterize the baseline system performance and accuracy of trained models. ImageNet is a large dataset that contains over 15 million labeled high-resolution images belonging to around 22,000 different categories [8]. The images were gathered from a variety of sources on the web and labeled by humans using Mechanical Turk. ImageNet contains images with variable resolution but like others we down-sampled all images to a fixed 256x256 resolution and used half of the data set for training and the other half for testing. This is the largest publicly available image classification benchmark and the task of correctly classifying an image among 22,000 categories is extremely hard (for e.g., distinguishing between an American and English foxhound). Performance on this task is measured in terms of top-1 accuracy, which compares the model's top choice with the image label and assigns a score of 1 for a correct answer and 0 for an incorrect answer. No partial credit is awarded. Random guessing will result in a top-1 accuracy of only around 0.0045%. Based on our experience with this benchmark it is unlikely that human performance exceeds 20% accuracy as this task requires correctly distinguishing between hundreds of breeds of dogs, butterflies, flowers, etc.[1] We use this benchmark to characterize Adam's performance and scaling, and the accuracy of trained models.

4.2 System Hardware

Adam is currently comprised of a cluster of 120 identical machines organized as three equally sized racks connected by IBM G8264 switches. Each machine is a HP Proliant server with dual Intel Xeon E5-2450L processors for a total of 16 cores running at 1.8Ghz with 98GB of main memory, two 10 Gb NICs and one 1 Gb NIC. All machines have four 7200 rpm HDDs. A 1TB drive hosts the operating system (Windows 2012 server) and the other three HDDs are 3TB each and are configured as a RAID array. This set of machines can be configured slightly differently based on the experiment but model training machines are selected from a pool of 90 machines, parameter servers from a pool of 20 machines and image servers from a pool of 10 machines. These pools include standby machines for fault tolerance in case of machine failure.

4.3 Baseline Performance and Accuracy

We first evaluate Adam's baseline performance by focusing on single model training and parameter server machines. In addition, we evaluate baseline training accuracy by training a small model on the MNIST digit classification task.

[1] We invite people to test their performance on this benchmark available at http://www.image-net.org

4.3.1 Model Training System

We train a small MNIST model comprising around 2.5 million connections (described later) to convergence on a single model training machine with no parameter server and vary the number of processor cores used for training. We measure the average training speed computed as billions of connections trained per second (*Model connections*Training examples*Number of Epochs)/(Wall clock time*) and plot this against the number of processor cores used for training. The results are shown in Figure 8. Adam shows excellent scaling as we increase the number of cores since we allow parameters to be updated without locking. The scaling is super-linear up to 4 cores due to caching effects and linear afterwards.

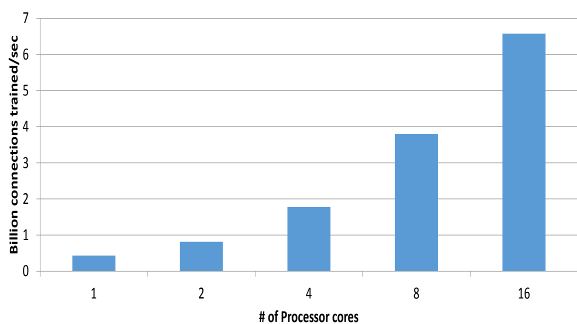


Figure 8. Model Training Node Performance.

4.3.2 Parameter Server

To evaluate the multi-core scaling of a single parameter server we collected parameter update traffic from ImageNet 22K model training runs, as MNIST parameter updates are too small to stress the system, and ran a series of simulated tests. For all tests we compare the parameter update rate that the machine is able to sustain as we increase the amount of server cores available for processing. Recall that we support two update APIs—one where the parameter server directly receives weight updates and the other where it receives activation and error gradient vectors that it must multiply to compute the weight updates. The results are shown in Figure 9. The network bandwidth is the limiting factor when weight updates are sent over the network resulting in poor performance and scaling. With a hypothetical fast network we see scaling up to 8 cores after which we hit the memory bandwidth bottleneck. When the weight updates are computed locally we see good scaling as we have tiled the computation to efficiently use the processor cache avoiding the memory bandwidth bottleneck. While our current networking technology limits our update throughput, we are still able to sustain a very high update rate of over 13 Bn updates/sec.

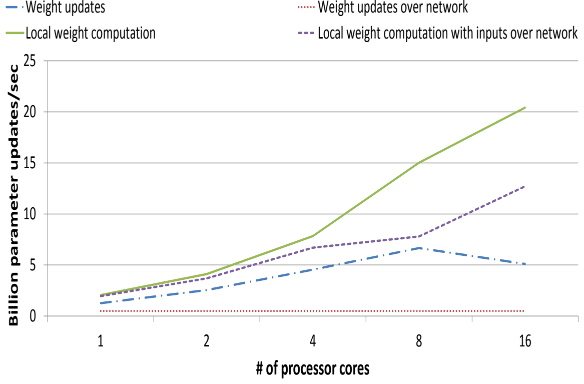


Figure 9. Parameter Server Node Performance.

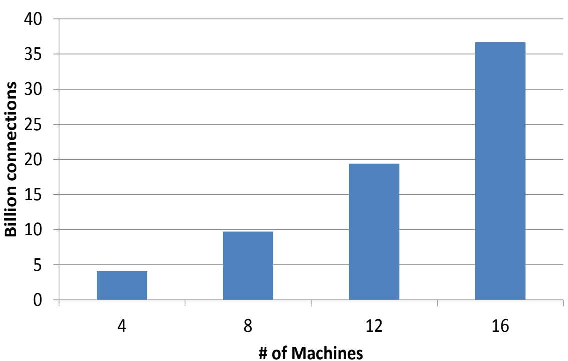


Figure 10. Scaling Model Size with more Workers.

4.3.3 Trained Model Accuracy

The MNIST benchmark is primarily evaluated in two forms. One variant transforms the training data via affine transformations or elastic distortions to effectively expand the limited training data to a much larger set resulting in the trained models generalizing well and achieving higher accuracy on the unseen test data [5, 26]. The traditional form allows no data transformation so all training has to proceed using only the limited 60,000 training examples. Since our goal here is to evaluate Adam's baseline performance on small models trained on little data we used the MNIST data without any transformation.

We trained a fairly standard model for this benchmark comprising 2 convolutional layers followed by two fully connected layers and a final ten class softmax output layer [26]. The convolutional layers used 5x5 kernels and each is followed by a 2x2 max-pooling layer. The first convolutional layer has 10 feature maps and the second has 20. Both fully connected layers use 400 hidden units. The resulting model is small and has around 2.5 million connections. The prediction accuracy results are shown in Table 1. We were

targeting competitive performance with the state-of-the-art accuracy on this benchmark from Goodfellow et al. that uses sophisticated training techniques that we have not implemented [12]. To our surprise, we exceeded their accuracy by 0.08%. To put this improvement in perspective, it took four years of advances in deep learning to improve accuracy on this task by 0.08% to its present value. We believe that our accuracy improvement arises from the asynchrony in Adam which adds a form of stochastic noise while training that helps the models generalize better when presented with unseen data. In addition, it is possible that the asynchrony helps the model escape from unstable local minima to potentially find a better local minimum. To validate this hypothesis, we trained the same model on the MNIST data using only a single thread to ensure synchronous training. We trained the model to convergence, which took significantly longer. The result from our best synchronous variant is shown in Table 1 and indicates that asynchrony contributes to improving model accuracy by 0.24%, which is a significant increase for this task. This result contradicts conventional established wisdom in the field that holds that asynchrony lowers model prediction accuracy and must be controlled as far as possible.

Table 1. MNIST Top-1 Accuracy

Systems	MNIST Top-1 Accuracy
Goodfellow et al [12]	99.55%
Adam	**99.63%**
Adam (synchronous)	99.39%

4.4 System Scaling and Accuracy

We evaluate our system performance and scalability across multiple dimensions and evaluate its ability to train large DNNs for the ImageNet 22K classification task.

4.4.1 Scaling with Model Workers

We evaluate the ability of Adam to train very large models by partitioning them across multiple machines. We use a single training epoch of the ImageNet benchmark to determine the maximum size model we can efficiently train on a given multi-machine configuration. We do this by increasing the model size via an increase in the number of feature maps in convolutional layers and training the model for an epoch until we observe a decrease in training speed. For this test we use only a single model replica with no parameter server. The results are shown in Fig. 10 and indicate that Adam is capable of training extremely large models using a relatively small number of machines. Our 16 machine configuration is capable of training a 36 Bn connection model. More importantly, the size of models we can train efficiently increases super-linearly as we partition the model across more machines. Our measurements indicate that this is due to cache effects where larger portions of the working sets of model layers fits in the L3 cache as the number of machines is increased. While the ImageNet data set does not have sufficient training data to train such large models to convergence these results indicate that Adam is capable of training very large models with good scaling.

4.4.2 Scaling with Model Replicas

We evaluate the impact of adding more model replicas to Adam. Each replica contains 4 machines with the ImageNet model (described later) partitioned across these machines. The results are shown in Figure 11 where we evaluated configurations comprising 4, 10, 12, 16, and 22 replicas. All experiments used the same parameter server configuration comprised of 20 machines. The results indicate that Adam scales well with additional replicas. Note that the configuration without a parameter server is merely intended as a reference for comparison since the models cannot jointly learn without a shared parameter server. While the parameter server does add some overhead the system still exhibits good scaling.

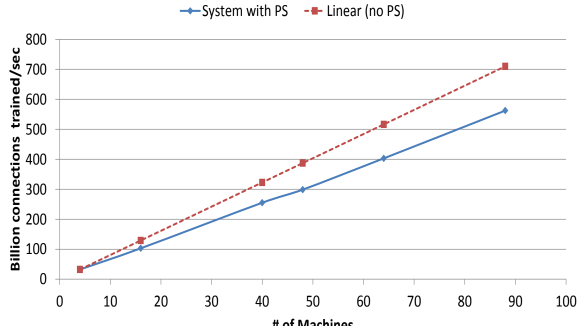


Figure 11. System scaling with more Replicas.

4.4.3 Trained Model Accuracy

We trained a large and deep convolutional network for the ImageNet 22K category object classification task with a similar architecture to those described in prior work [17, 27]. The network has five convolutional layers followed by three fully connected layers with a

Table 2. ImageNet 22K Top-1 Accuracy.

Systems	ImageNet 22K Top-1 Accuracy
Le et al. [18]	13.6%
Le et al. (with pre-training) [18]	15.8%
Adam	**29.8%**

final 22,000-way softmax. The convolutional kernels range in size from 3x3 to 7x7 and the convolutional feature map sizes range from 120 to 600. The first, second and fifth convolutional layers are followed by a 3x3 max-pooling layer. The fully-connected layers contain 3000 hidden units. The resulting model is fairly large and contains over 2Bn connections. While Adam is capable of training much larger models the amount of ImageNet training data is a limiting factor in these experiments.

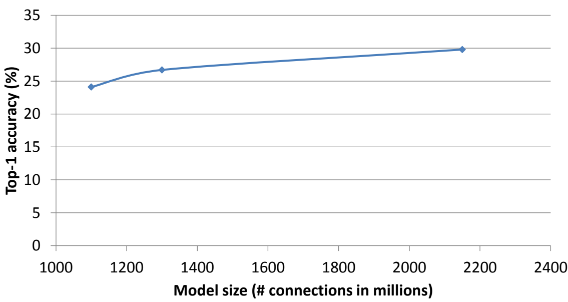


Figure 12. Model accuracy with larger models.

We trained this model to convergence in ten days using 4 image servers, 48 model training machines configured as 16 model replicas containing 4 machines per replica and 10 parameter servers for a total of 62 machines. The results are shown in Table 2. Le et al. held the previous best top-1 accuracy result on this benchmark of 13.6% that was obtained by training a 1Bn connection model on 2,000 machines for a week (our model exceeds 13.6% accuracy with a single day of training using 62 machines). When they supplemented the ImageNet training data with 10 million unlabeled images sampled from Youtube videos, which they trained on using 1,000 machines for 3 days, they were able to increase prediction accuracy to 15.8%. Our model is able to achieve a new world record prediction accuracy of 29.8% using only ImageNet training data, which is a dramatic 2x improvement over the prior best.

To better understand the reasons for this accuracy improvement, we used Adam to train a couple of smaller models to convergence for this task. The results are shown in Figure 12 and indicate that training larger models increases task accuracy. This highlights the importance of Adam's efficiency and scalability as it enables training larger models. In addition, our 1.1 Bn connection model achieves 24% accuracy on this task as compared to prior work that achieved 13.6% accuracy with a similar size model. While we are unable to isolate the impact of asynchrony for this task as the synchronous execution is much too slow, this result in conjunction with the MNIST accuracy data provides evidence that asynchrony contributes to the accuracy improvements. The graph also appears to suggest that improvements in accuracy slow down as the model size increases but we note that the larger models are being trained with the same amount of data. It is likely that larger models for complex tasks require more training data to effectively use their capacity.

4.4.4 Discussion

Adam achieves high multi-threaded scalability on a single machine by permitting threads to update local parameter weights without locks. It achieves good multi-machine scalability through minimizing communication traffic by performing the weight update computation on the parameter server machines and performing asynchronous batched updates to parameter values that take advantage of these updates being associative and commutative. Finally, Adam enables training models to high accuracy by exploiting its efficiency to train very large models and leveraging asynchrony to further improve accuracy.

5. RELATED WORK

Due to the computational requirements of deep learning, deep models are popularly trained on GPUs [5, 14, 17, 24, 27]. While this works well when the model fits within 2-4 GPU cards attached to a single server, it limits the size of models that can be trained. Consequently the models trained on these systems are typically evaluated on the much smaller ImageNet 1,000 category classification task [17, 27].

Recent work attempted to use a distributed cluster of 16 GPU servers connected with Infiniband to train large DNNs partitioned across the servers on image classification tasks [6]. Training large models to high accuracy typically requires iterating over vast amount of data. This is not viable in a reasonable amount of time unless the system also supports data parallelism. Unfortunately the mismatch in speed between GPU compute and network interconnects makes it extremely difficult to support data parallelism via a parameter server. Either the GPU must constantly stall while waiting for model parameter updates or the models will likely diverge due to insufficient synchronization. This work did not support data parallelism and the large models trained had lower accuracy than much smaller models.

The only comparable system that we are aware of for training large-scale DNNs that supports both model and data parallelism is the DistBelief system [7]. The system has been used to train a large DNN (1 billion connections) to high accuracy on the ImageNet 22K classification task but at a significant compute cost of using 2,000 machines for a week. In addition, the

system exhibits poor scaling efficiency and is not a viable cost-effective solution.

GraphLab [21] and similar large scale graph processing frameworks are designed for operating on general unstructured graphs and are unlikely to offer competitive performance and scalability as they do not exploit deep network structure and training efficiencies.

The vision and computer architecture community has started to explore hardware acceleration for neural network models for vision [3, 4, 10, 16, 22]. Currently, the work has concentrated on efficient feed-forward evaluation of already trained networks and complements our work that focuses on training large DNNs.

6. CONCLUSIONS

We show that large-scale commodity distributed systems can be used to efficiently train very large DNNs to world-record accuracy on hard vision tasks using current training algorithms by using Adam to train a large DNN model that achieves world-record classification performance on the ImageNet 22K category task. While we have implemented and evaluated Adam using a 120 machine cluster, the scaling results indicate that much larger systems can likely be effectively utilized for training large DNNs.

7. ACKNOWLEDGMENTS

We would like to thank Patrice Simard for sharing his gradient descent toolkit code that we started with as a single machine reference implementation. Leon Bottou provided valuable guidance and advice on scalable training algorithms. John Platt served as our machine learning consultant throughout this effort and constantly shared valuable input. Yi-Min Wang was an early and constant supporter of this work and provided the initial seed funding. Peter Lee and Eric Horvitz provided additional support and funding. Jim Larus, Eric Rudder and Qi Lu encouraged this work. We would also like to acknowledge the contributions of Olatunji Ruwase, Abhishek Sharma, and Xinying Song to the system. We benefitted from several discussions with Larry Zitnick, Piotr Dollar, Istvan Cseri, Slavik Krassovsky, and Sven Groot. Finally, we would like to thank our reviewers and our paper shepherd, Geoff Voelker, for their detailed and thoughtful comments.

8. REFERENCES

[1] Bengio, Y., and LeCun, Y. 2007. Scaling Learning Algorithms towards AI. In *Large-Scale Kernel Machines*, Bottou, L. et al. (Eds), MIT Press.

[2] Bottou, L., 1991. Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nîmes 91*, EC2, Nimes, France.

[3] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *International symposium on Computer Architecture*, ISCA'10.

[4] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS'14.

[5] Ciresan, D. C, Meier, U., and Schmidhuber, J. 2012. Multicolumn deep neural networks for image classification. In *Computer Vision and Pattern Recognition*. CVPR'12.

[6] Coates, A., Huval, B., Wang, T., Wu, D., Ng, A., and Catanzaro, B. 2013. Deep Learning with COTS HPC. In *International Conference on Machine Learning*. ICML'13.

[7] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q., and Ng, A. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*. NIPS'12.

[8] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *Computer Vision and Pattern Recognition*. CVPR '09.

[9] Faerber, P., and Asanović, K. 1997. Parallel neural network training on Multi-Spert. In *IEEE 3rd International Conference on Algorithms and Architectures for Parallel Processing* (Melbourne, Australia, December 1997).

[10] Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., and LeCun, Y. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition* Workshop (June 2011), pages 109–116.

[11] Fukushima, K. 1980. Neocognitron: A self-organizing neural network for a mechanism of pattern recognition unaffected by shift in position. In *Biological Cybernetics*, 36(4): 93-202.

[12] Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. 2013. Maxout

Networks. In *International Conference on Machine Learning*. ICML'13.

[13] Hahnloser, R. 2003. Permitted and Forbidden Sets in Symmetric Threshold-Linear Networks. In *Neural Computing*. (Mar. 2003), 15(3):621-38.

[14] Hinton, G., Deng, L., Yu, D., Dahl, G., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. 2012. Deep neural networks for acoustic modeling in speech recognition. In *IEEE Signal Processing Magazine*, 2012.

[15] Hubel, D. and Wiesel, T. 1968. Receptive fields and functional architecture of monkey striate cortex. In *Journal of Physiology* (London), 195, 215–243.

[16] Kim, J., Member, S., Kim, M., Lee, S., Oh, J., Kim, K. and Yoo, H. 2010. A 201.4 GOPS 496 mW Real-Time Multi-Object Recognition Processor with Bio-Inspired Neural Perception Engine. In *IEEE Journal of Solid-State Circuits*, (Jan. 2010), 45(1):32–45.

[17] Krizhevsky, A., Sutskever, I., and Hinton, G. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. NIPS'12.

[18] Le, Q., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G., Dean, J., and Ng, A. 2012. Building high-level features using large scale unsupervised learning. In *International Conference on Machine Learning*. ICML'12.

[19] LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., and Jackel, L. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. In *Neural Computation*, 1(4):541-551.

[20] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 86(11):2278–2324, (Nov. 1998).

[21] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. 2012. Distributed GraphLab: A framework for machine learning in the cloud. In *International Conference on Very Large Databases.* VLDB'12.

[22] Maashri, A., Debole, M., Cotter, M., Chandramoorthy, N., Xiao, Y., Narayanan, V., and Chakrabarti, C. 2012. Accelerating neuromorphic vision algorithms for recognition. In *Proceedings of the 49th Annual Design Automation Conference*, DAC'12.

[23] Niu, F., Retcht, B., Re, C., and Wright, S. 2011. Hogwild! A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*. NIPS'11.

[24] Raina, R., Madhavan, A., and Ng., A. 2009. Large-scale deep unsupervised learning using graphics processors. In *International Conference on Machine Learning.* ICML'09.

[25] Rumelhart, D., Hinton, G., and Williams, R. 1986. Learning representations by back-propagating errors. In *Nature* 323 (6088): 533–536.

[26] Simard, P., Steinkraus, D., and Platt, J. 2003. Best Practices for Convolutional Neural Networks applied to Visual Document Analysis. In *ICDAR*, vol. 3, pp. 958-962.

[27] Zeiler, M. and Fergus, R. 2013. Visualizing and Understanding Convolutional Networks. In *Arxiv 1311.2901*. http://arxiv.org/abs/1311.2901

Scaling Distributed Machine Learning with the Parameter Server

Mu Li[*‡], David G. Andersen[*], Jun Woo Park[*], Alexander J. Smola[*†], Amr Ahmed[†], Vanja Josifovski[†], James Long[†], Eugene J. Shekita[†], Bor-Yiing Su[†]

[*]Carnegie Mellon University [‡]Baidu [†]Google

{muli, dga, junwoop}@cs.cmu.edu, alex@smola.org, {amra, vanjaj, jamlong, shekita, boryiingsu}@google.com

Abstract

We propose a parameter server framework for distributed machine learning problems. Both data and workloads are distributed over worker nodes, while the server nodes maintain globally shared parameters, represented as dense or sparse vectors and matrices. The framework manages asynchronous data communication between nodes, and supports flexible consistency models, elastic scalability, and continuous fault tolerance.

To demonstrate the scalability of the proposed framework, we show experimental results on petabytes of real data with billions of examples and parameters on problems ranging from Sparse Logistic Regression to Latent Dirichlet Allocation and Distributed Sketching.

1 Introduction

Distributed optimization and inference is becoming a prerequisite for solving large scale machine learning problems. At scale, no single machine can solve these problems sufficiently rapidly, due to the growth of data and the resulting model complexity, often manifesting itself in an increased number of parameters. Implementing an efficient distributed algorithm, however, is not easy. Both intensive computational workloads and the volume of data communication demand careful system design.

Realistic quantities of training data can range between 1TB and 1PB. This allows one to create powerful and complex models with 10^9 to 10^{12} parameters [9]. These models are often shared globally by all worker nodes, which must frequently accesses the shared parameters as they perform computation to refine it. Sharing imposes three challenges:

- Accessing the parameters requires an enormous amount of network bandwidth.
- Many machine learning algorithms are sequential. The resulting barriers hurt performance when the cost of synchronization and machine latency is high.
- At scale, fault tolerance is critical. Learning tasks are often performed in a cloud environment where machines can be unreliable and jobs can be preempted.

≈ #machine × time	# of jobs	failure rate
100 hours	13,187	7.8%
1,000 hours	1,366	13.7%
10,000 hours	77	24.7%

Table 1: Statistics of machine learning jobs for a three month period in a data center.

To illustrate the last point, we collected all job logs for a three month period from one cluster at a large internet company. We show statistics of batch machine learning tasks serving a production environment in Table 1. Here, task failure is mostly due to being preempted or losing machines without necessary fault tolerance mechanisms.

Unlike in many research settings where jobs run exclusively on a cluster without contention, fault tolerance is a necessity in real world deployments.

1.1 Contributions

Since its introduction, the parameter server framework [43] has proliferated in academia and industry. This paper describes a third generation open source implementation of a parameter server that focuses on the systems aspects of distributed inference. It confers two advantages to developers: First, by factoring out commonly required components of machine learning systems, it enables application-specific code to remain concise. At the same time, as a shared platform to target for systems-level optimizations, it provides a robust, versatile, and high-performance implementation capable of handling a diverse array of algorithms from sparse logistic regression to topic models and distributed sketching. Our design de-

	Shared Data	Consistency	Fault Tolerance
Graphlab [34]	graph	eventual	checkpoint
Petuum [12]	hash table	delay bound	none
REEF [10]	array	BSP	checkpoint
Naiad [37]	(key,value)	multiple	checkpoint
Mlbase [29]	table	BSP	RDD
Parameter Server	(sparse) vector/matrix	various	continuous

Table 2: Attributes of distributed data analysis systems.

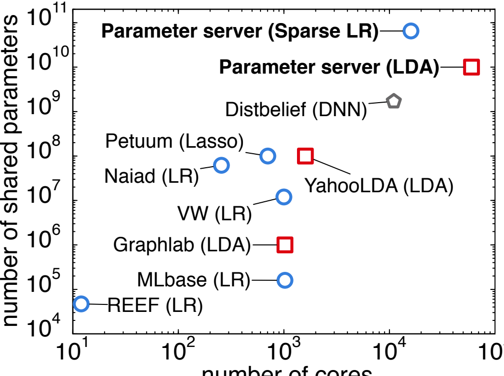


Figure 1: Comparison of the public largest machine learning experiments each system performed. Problems are color-coded as follows: Blue circles — sparse logistic regression; red squares — latent variable graphical models; grey pentagons — deep networks.

cisions were guided by the workloads found in real systems. Our parameter server provides five key features:

Efficient communication: The asynchronous communication model does not block computation (unless requested). It is optimized for machine learning tasks to reduce network traffic and overhead.

Flexible consistency models: Relaxed consistency further hides synchronization cost and latency. We allow the algorithm designer to balance algorithmic convergence rate and system efficiency. The best trade-off depends on data, algorithm, and hardware.

Elastic Scalability: New nodes can be added *without* restarting the running framework.

Fault Tolerance and Durability: Recovery from and repair of non-catastrophic machine failures within 1s, without interrupting computation. Vector clocks ensure well-defined behavior after network partition and failure.

Ease of Use: The globally shared parameters are represented as (potentially sparse) vectors and matrices to facilitate development of machine learning applications. The linear algebra data types come with high-performance multi-threaded libraries.

The novelty of the proposed system lies in the synergy achieved by picking the right systems techniques, adapting them to the machine learning algorithms, and modifying the machine learning algorithms to be more systems-friendly. In particular, we can relax a number of otherwise hard systems constraints since the associated machine learning algorithms are quite tolerant to perturbations. The consequence is the first general purpose ML system capable of scaling to industrial scale sizes.

1.2 Engineering Challenges

When solving distributed data analysis problems, the issue of reading and updating parameters shared between different worker nodes is ubiquitous. The parameter server framework provides an efficient mechanism for aggregating and synchronizing model parameters and statistics between workers. Each parameter server node maintains only a part of the parameters, and each worker node typically requires only a subset of these parameters when operating. Two key challenges arise in constructing a high performance parameter server system:

Communication. While the parameters could be updated as key-value pairs in a conventional datastore, using this abstraction naively is inefficient: values are typically small (floats or integers), and the overhead of sending each update as a key value operation is high.

Our insight to improve this situation comes from the observation that many learning algorithms represent parameters as structured mathematical objects, such as vectors, matrices, or tensors. At each logical time (or an iteration), typically a part of the object is updated. That is, workers usually send a *segment* of a vector, or an entire *row* of the matrix. This provides an opportunity to automatically batch both the communication of updates and their processing on the parameter server, and allows the consistency tracking to be implemented efficiently.

Fault tolerance, as noted earlier, is critical at scale, and for efficient operation, it must not require a full restart of a long-running computation. Live replication of parameters between servers supports hot failover. Failover and self-repair in turn support dynamic scaling by treating machine removal or addition as failure or repair respectively.

Figure 1 provides an overview of the scale of the largest supervised and unsupervised machine learning experiments performed on a number of systems. When possible, we confirmed the scaling limits with the authors of each of these systems (data current as of 4/2014). As is evident, we are able to cover orders of magnitude more data on orders of magnitude more processors than any

other published system. Furthermore, Table 2 provides an overview of the main characteristics of several machine learning systems. Our parameter server offers the greatest degree of flexibility in terms of consistency. It is the only system offering continuous fault tolerance. Its native data types make it particularly friendly for data analysis.

1.3 Related Work

Related systems have been implemented at Amazon, Baidu, Facebook, Google [13], Microsoft, and Yahoo [1]. Open source codes also exist, such as YahooLDA [1] and Petuum [24]. Furthermore, Graphlab [34] supports parameter synchronization on a best effort model.

The first generation of such parameter servers, as introduced by [43], lacked flexibility and performance — it repurposed `memcached` distributed (key,value) store as synchronization mechanism. YahooLDA improved this design by implementing a dedicated server with user-definable update primitives (set, get, update) and a more principled load distribution algorithm [1]. This second generation of *application specific* parameter servers can also be found in Distbelief [13] and the synchronization mechanism of [33]. A first step towards a general platform was undertaken by Petuum [24]. It improves YahooLDA with a bounded delay model while placing further constraints on the worker threading model. We describe a third generation system overcoming these limitations.

Finally, it is useful to compare the parameter server to more general-purpose distributed systems for machine learning. Several of them mandate synchronous, iterative communication. They scale well to tens of nodes, but at large scale, this synchrony creates challenges as the chance of a node operating slowly increases. Mahout [4], based on Hadoop [18] and MLI [44], based on Spark [50], both adopt the iterative MapReduce [14] framework. A key insight of Spark and MLI is preserving state between iterations, which is a core goal of the parameter server.

Distributed GraphLab [34] instead asynchronously schedules communication using a graph abstraction. At present, GraphLab lacks the elastic scalability of the map/reduce-based frameworks, and it relies on coarse-grained snapshots for recovery, both of which impede scalability. Its applicability for certain algorithms is limited by its lack of global variable synchronization as an efficient first-class primitive. In a sense, a core goal of the parameter server framework is to capture the benefits of GraphLab's asynchrony without its structural limitations.

Piccolo [39] uses a strategy related to the parameter server to share and aggregate state between machines. In it, workres pre-aggregate state locally and transmit the updates to a server keeping the aggregate state. It thus implements largely a subset of the functionality of our system, lacking the mechane learning specailized optimizations: message compression, replication, and variable consistency models expressed via dependency graphs.

2 Machine Learning

Machine learning systems are widely used in Web search, spam detection, recommendation systems, computational advertising, and document analysis. These systems automatically learn models from examples, termed *training data*, and typically consist of three components: *feature extraction*, the *objective function*, and *learning*.

Feature extraction processes the raw training data, such as documents, images and user query logs, to obtain *feature vectors*, where each feature captures an attribute of the training data. Preprocessing can be executed efficiently by existing frameworks such as MapReduce, and is therefore outside the scope of this paper.

2.1 Goals

The goal of many machine learning algorithms can be expressed via an "objective function." This function captures the properties of the learned model, such as low error in the case of classifying e-mails into ham and spam, how well the data is explained in the context of estimating topics in documents, or a concise summary of counts in the context of sketching data.

The learning algorithm typically minimizes this objective function to obtain the model. In general, there is no closed-form solution; instead, learning starts from an initial model. It iteratively refines this model by processing the training data, possibly multiple times, to approach the solution. It stops when a (near) optimal solution is found or the model is considered to be converged.

The training data may be extremely large. For instance, a large internet company using one year of an ad impression log [27] to train an *ad click predictor* would have trillions of training examples. Each training example is typically represented as a possibly very high-dimensional "feature vector" [9]. Therefore, the training data may consist of trillions of trillion-length feature vectors. Iteratively processing such large scale data requires enormous computing and bandwidth resources. Moreover, billions of new ad impressions may arrive daily. Adding this data into the system often improves both prediction accuracy and coverage. But it also requires the learning algorithm to run daily [35], possibly in real time. Efficient execution of these algorithms is the main focus of this paper.

To motivate the design decisions in our system, next we briefly outline the two widely used machine learning technologies that we will use to demonstrate the efficacy of our parameter server. More detailed overviews can be found in [36, 28, 42, 22, 6].

2.2 Risk Minimization

The most intuitive variant of machine learning problems is that of risk minimization. The "risk" is, roughly, a measure of prediction error. For example, if we were to predict tomorrow's stock price, the risk might be the deviation between the prediction and the actual value of the stock.

The training data consists of n examples. x_i is the ith such example, and is often a vector of length d. As noted earlier, both n and d may be on the order of billions to trillions of examples and dimensions, respectively. In many cases, each training example x_i is associated with a label y_i. In ad click prediction, for example, y_i might be 1 for "clicked" or -1 for "not clicked".

Risk minimization learns a model that can predict the value y of a future example x. The model consists of parameters w. In the simplest example, the model parameters might be the "clickiness" of each feature in an ad impression. To predict whether a new impression would be clicked, the system might simply sum its "clickiness" based upon the features present in the impression, namely $x^\top w := \sum_{j=1}^{d} x_j w_j$, and then decide based on the sign.

In any learning algorithm, there is an important relationship between the amount of training data and the model size. A more detailed model typically improves accuracy, but only up to a point: If there is too little training data, a highly-detailed model will *overfit* and become merely a system that uniquely memorizes every item in the training set. On the other hand, a too-small model will fail to capture interesting and relevant attributes of the data that are important to making a correct decision.

Regularized risk minimization [48, 19] is a method to find a model that balances model complexity and training error. It does so by minimizing the sum of two terms: a *loss* $\ell(x, y, w)$ representing the prediction error on the training data and a *regularizer* $\Omega[w]$ penalizing the model complexity. A good model is one with low error and low complexity. Consequently we strive to minimize

$$F(w) = \sum_{i=1}^{n} \ell(x_i, y_i, w) + \Omega(w). \tag{1}$$

The specific loss and regularizer functions used are important to the prediction performance of the machine learning algorithm, but relatively unimportant for the purpose of this paper: the algorithms we present can be used with all of the most popular loss functions and regularizers.

In Section 5.1 we use a high-performance distributed learning algorithm to evaluate the parameter server. For the sake of simplicity we describe a much simpler model

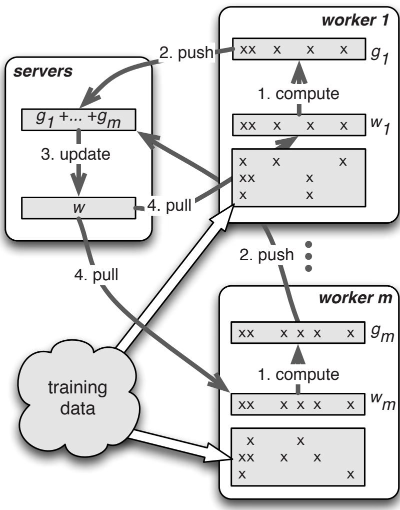


Figure 2: Steps required in performing distributed subgradient descent, as described e.g. in [46]. Each worker only caches the working set of w rather than all parameters.

Algorithm 1 Distributed Subgradient Descent

Task Scheduler:

1: issue LoadData() to all workers
2: **for** iteration $t = 0, \ldots, T$ **do**
3:     issue WORKERITERATE(t) to all workers.
4: **end for**

Worker $r = 1, \ldots, m$:

1: **function** LOADDATA()
2:     load a part of training data $\{y_{i_k}, x_{i_k}\}_{k=1}^{n_r}$
3:     pull the working set $w_r^{(0)}$ from servers
4: **end function**
5: **function** WORKERITERATE(t)
6:     gradient $g_r^{(t)} \leftarrow \sum_{k=1}^{n_r} \partial\ell(x_{i_k}, y_{i_k}, w_r^{(t)})$
7:     push $g_r^{(t)}$ to servers
8:     pull $w_r^{(t+1)}$ from servers
9: **end function**

Servers:

1: **function** SERVERITERATE(t)
2:     aggregate $g^{(t)} \leftarrow \sum_{r=1}^{m} g_r^{(t)}$
3:     $w^{(t+1)} \leftarrow w^{(t)} - \eta\left(g^{(t)} + \partial\Omega(w^{(t)})\right)$
4: **end function**

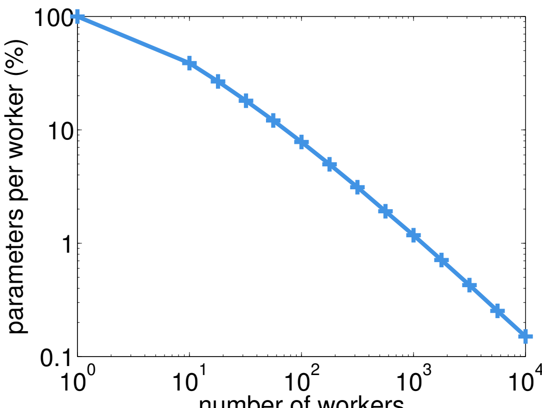


Figure 3: Each worker's set of parameters shrinks as more workers are used, requiring less memory per machine.

[46] called *distributed subgradient descent*.[1]

As shown in Figure 2 and Algorithm 1, the training data is partitioned among all of the workers, which jointly learn the parameter vector w. The algorithm operates iteratively. In each iteration, every worker independently uses its own training data to determine what changes should be made to w in order to get closer to an optimal value. Because each worker's updates reflect only its own training data, the system needs a mechanism to allow these updates to mix. It does so by expressing the updates as a subgradient—a direction in which the parameter vector w should be shifted—and aggregates all subgradients before applying them to w. These gradients are typically scaled down, with considerable attention paid in algorithm design to the right *learning rate* η that should be applied in order to ensure that the algorithm converges quickly.

The most expensive step in Algorithm 1 is computing the subgradient to update w. This task is divided among all of the workers, each of which execute WORKERITERATE. As part of this, workers compute $w^\top x_{i_k}$, which could be infeasible for very high-dimensional w. Fortunately, a worker needs to know a coordinate of w if and only if some of its training data references that entry.

For instance, in ad-click prediction one of the key features are the words in the ad. If only very few advertisements contain the phrase *OSDI 2014*, then most workers will not generate any updates to the corresponding entry in w, and hence do not require this entry. While the *total* size of w may exceed the capacity of a single machine, the working set of entries needed by a particular worker can be trivially cached locally. To illustrate this, we randomly assigned data to workers and then counted the average working set size per worker on the dataset that is used in Section 5.1. Figure 3 shows that for 100 workers, each worker only needs 7.8% of the total parameters. With 10,000 workers this reduces to 0.15%.

2.3 Generative Models

In a second major class of machine learning algorithms, the label to be applied to training examples is unknown. Such settings call for *unsupervised* algorithms (for labeled training data one can use *supervised* or *semi-supervised* algorithms). They attempt to capture the underlying structure of the data. For example, a common problem in this area is *topic modeling*: Given a collection of documents, infer the topics contained in each document.

When run on, e.g., the SOSP'13 proceedings, an algorithm might generate topics such as "distributed systems", "machine learning", and "performance." The algorithms infer these topics from the content of the documents themselves, not an external topic list. In practical settings such as content personalization for recommendation systems [2], the scale of these problems is huge: hundreds of millions of users and billions of documents, making it critical to parallelize the algorithms across large clusters.

Because of their scale and data volumes, these algorithms only became commercially applicable following the introduction of the first-generation parameter servers [43]. A key challenge in topic models is that the parameters describing the current estimate of how documents are supposed to be generated must be shared.

A popular topic modeling approach is Latent Dirichlet Allocation (LDA) [7]. While the statistical model is quite different, the resulting algorithm for learning it is very similar to Algorithm 1.[2] The key difference, however, is that the update step is not a gradient computation, but an estimate of how well the document can be explained by the current model. This computation requires access to auxiliary metadata for each document that is updated each time a document is accessed. Because of the number of documents, metadata is typically read from and written back to disk whenever the document is processed.

This auxiliary data is the set of topics assigned to each word of a document, and the parameter w being learned consists of the relative frequency of occurrence of a word.

As before, each worker needs to store only the parameters for the words occurring in the documents it processes. Hence, distributing documents across workers has

[1] The unfamiliar reader could read this as *gradient descent*; the subgradient aspect is simply a generalization to loss functions and regularizers that need not be continuously differentiable, such as $|w|$ at $w = 0$.

[2] The specific algorithm we use in the evaluation is a parallelized variant of a stochastic variational sampler [25] with an update strategy similar to that used in YahooLDA [1].

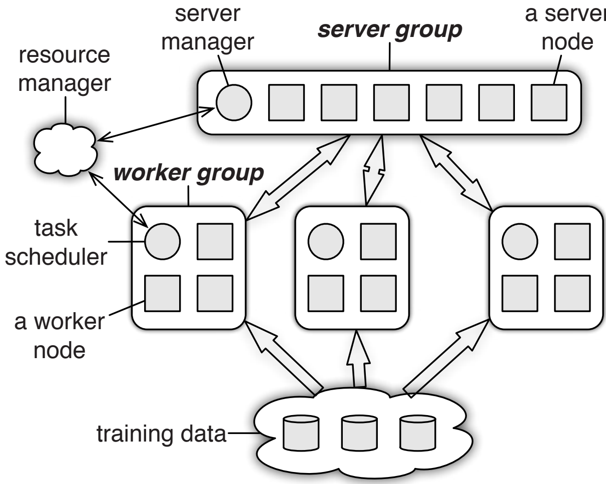


Figure 4: Architecture of a parameter server communicating with several groups of workers.

the same effect as in the previous section: we can process much bigger models than a single worker may hold.

3 Architecture

An instance of the parameter server can run more than one algorithm simultaneously. Parameter server nodes are grouped into a server group and several worker groups as shown in Figure 4. A server node in the server group maintains a partition of the globally shared parameters. Server nodes communicate with each other to replicate and/or to migrate parameters for reliability and scaling. A server manager node maintains a consistent view of the metadata of the servers, such as node liveness and the assignment of parameter partitions.

Each worker group runs an application. A worker typically stores locally a portion of the training data to compute local statistics such as gradients. Workers communicate only with the server nodes (not among themselves), updating and retrieving the shared parameters. There is a scheduler node for each worker group. It assigns tasks to workers and monitors their progress. If workers are added or removed, it reschedules unfinished tasks.

The parameter server supports independent parameter namespaces. This allows a worker group to isolate its set of shared parameters from others. Several worker groups may also share the same namespace: we may use more than one worker group to solve the same deep learning application [13] to increase parallelization. Another example is that of a model being actively queried by some nodes, such as online services consuming this model. Simultaneously the model is updated by a different group of worker nodes as new training data arrives.

The parameter server is designed to simplify developing distributed machine learning applications such as those discussed in Section 2. The shared parameters are presented as (key,value) vectors to facilitate linear algebra operations (Sec. 3.1). They are distributed across a group of server nodes (Sec. 4.3). Any node can both push out its local parameters and pull parameters from remote nodes (Sec. 3.2). By default, workloads, or tasks, are executed by worker nodes; however, they can also be assigned to server nodes via user defined functions (Sec. 3.3). Tasks are asynchronous and run in parallel (Sec. 3.4). The parameter server provides the algorithm designer with flexibility in choosing a consistency model via the task dependency graph (Sec. 3.5) and predicates to communicate a subset of parameters (Sec. 3.6).

3.1 (Key,Value) Vectors

The model shared among nodes can be represented as a set of (key, value) pairs. For example, in a loss minimization problem, the pair is a feature ID and its weight. For LDA, the pair is a combination of the word ID and topic ID, and a count. Each entry of the model can be read and written locally or remotely by its key. This (key,value) abstraction is widely adopted by existing approaches [37, 29, 12].

Our parameter server improves upon this basic approach by acknowledging the underlying meaning of these key value items: machine learning algorithms typically treat the model as a linear algebra object. For instance, w is used as a vector for both the objective function (1) and the optimization in Algorithm 1 by risk minimization. By treating these objects as sparse linear algebra objects, the parameter server can provide the same functionality as the (key,value) abstraction, but admits important optimized operations such as vector addition $w + u$, multiplication Xw, finding the 2-norm $\|w\|_2$, and other more sophisticated operations [16].

To support these optimizations, we assume that the keys are ordered. This lets us treat the parameters as (key,value) pairs while endowing them with vector and matrix semantics, where non-existing keys are associated with zeros. This helps with linear algebra in machine learning. It reduces the programming effort to implement optimization algorithms. Beyond convenience, this interface design leads to efficient code by leveraging CPU-efficient multithreaded self-tuning linear algebra libraries such as BLAS [16], LAPACK [3], and ATLAS [49].

3.2 Range Push and Pull

Data is sent between nodes using `push` and `pull` operations. In Algorithm 1 each worker pushes its entire local gradient into the servers, and then pulls the updated weight back. The more advanced algorithm described in Algorithm 3 uses the same pattern, except that only a range of keys is communicated each time.

The parameter server optimizes these updates for programmer convenience as well as computational and network bandwidth efficiency by supporting *range-based* push and pull. If $\mathcal{R}$ is a key range, then `w.push`$(\mathcal{R},$ `dest`$)$ sends all existing entries of w in key range $\mathcal{R}$ to the destination, which can be either a particular node, or a node group such as the server group. Similarly, `w.pull`$(\mathcal{R},$ `dest`$)$ reads all existing entries of w in key range $\mathcal{R}$ from the destination. If we set $\mathcal{R}$ to be the whole key range, then the whole vector w will be communicated. If we set $\mathcal{R}$ to include a single key, then only an individual entry will be sent.

This interface can be extended to communicate any local data structures that share the same keys as w. For example, in Algorithm 1, a worker pushes its temporary local gradient g to the parameter server for aggregation. One option is to make g globally shared. However, note that g shares the keys of the worker's working set w. Hence the programmer can use `w.push`$(\mathcal{R},$ `g`, `dest`$)$ for the local gradients to save memory and also enjoy the optimization discussed in the following sections.

3.3 User-Defined Functions on the Server

Beyond aggregating data from workers, server nodes can execute user-defined functions. It is beneficial because the server nodes often have more complete or up-to-date information about the shared parameters. In Algorithm 1, server nodes evaluate subgradients of the regularizer Ω in order to update w. At the same time a more complicated proximal operator is solved by the servers to update the model in Algorithm 3. In the context of sketching (Sec. 5.3), almost all operations occur on the server side.

3.4 Asynchronous Tasks and Dependency

A tasks is issued by a remote procedure call. It can be a `push` or a `pull` that a worker issues to servers. It can also be a user-defined function that the scheduler issues to any node. Tasks may include any number of subtasks. For example, the task `WorkerIterate` in Algorithm 1 contains one `push` and one `pull`.

Tasks are executed asynchronously: the caller can perform further computation immediately after issuing a task.

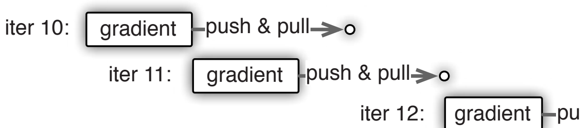


Figure 5: Iteration 12 depends on 11, while 10 and 11 are independent, thus allowing asynchronous processing.

The caller marks a task as finished only once it receives the callee's reply. A reply could be the function return of a user-defined function, the (key,value) pairs requested by the `pull`, or an empty acknowledgement. The callee marks a task as finished only if the call of the task is returned and all subtasks issued by this call are finished.

By default, callees execute tasks in parallel, for best performance. A caller that wishes to serialize task execution can place an *execute-after-finished* dependency between tasks. Figure 5 depicts three example iterations of `WorkerIterate`. Iterations 10 and 11 are independent, but 12 depends on 11. The callee therefore begins iteration 11 immediately after the local gradients are computed in iteration 10. Iteration 12, however, is postponed until the `pull` of 11 finishes.

Task dependencies help implement algorithm logic. For example, the aggregation logic in `ServerIterate` of Algorithm 1 updates the weight w only after all worker gradients have been aggregated. This can be implemented by having the updating task depend on the push tasks of all workers. The second important use of dependencies is to support the flexible consistency models described next.

3.5 Flexible Consistency

Independent tasks improve system efficiency via parallelizing the use of CPU, disk and network bandwidth. However, this may lead to data inconsistency between nodes. In the diagram above, the worker r starts iteration 11 before $w^{(11)}$ has been pulled back, so it uses the old $w_r^{(10)}$ in this iteration and thus obtains the same gradient as in iteration 10, namely $g_r^{(11)} = g_r^{(10)}$. This inconsistency potentially slows down the convergence progress of Algorithm 1. However, some algorithms may be less sensitive to this type of inconsistency. For example, only a segment of w is updated each time in Algorithm 3. Hence, starting iteration 11 without waiting for 10 causes only a part of w to be inconsistent.

The best trade-off between system efficiency and algorithm convergence rate usually depends on a variety of factors, including the algorithm's sensitivity to data inconsistency, feature correlation in training data, and capacity

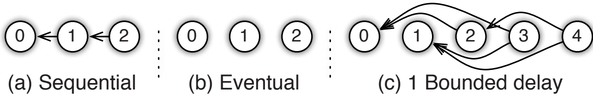


Figure 6: Directed acyclic graphs for different consistency models. The size of the DAG increases with the delay.

difference of hardware components. Instead of forcing the user to adopt one particular dependency that may be ill-suited to the problem, the parameter server gives the algorithm designer flexibility in defining consistency models. This is a substantial difference to other machine learning systems.

We show three different models that can be implemented by task dependency. Their associated directed acyclic graphs are given in Figure 6.

Sequential In sequential consistency, all tasks are executed one by one. The next task can be started only if the previous one has finished. It produces results identical to the single-thread implementation, and also named Bulk Synchronous Processing.

Eventual Eventual consistency is the opposite: all tasks may be started simultaneously. For instance, [43] describes such a system. However, this is only recommendable if the underlying algorithms are robust with regard to delays.

Bounded Delay When a maximal delay time τ is set, a new task will be blocked until all previous tasks τ times ago have been finished. Algorithm 3 uses such a model. This model provides more flexible controls than the previous two: $\tau = 0$ is the sequential consistency model, and an infinite delay $\tau = \infty$ becomes the eventual consistency model.

Note that the dependency graphs may be dynamic. For instance the scheduler may increase or decrease the maximal delay according to the runtime progress to balance system efficiency and convergence of the underlying optimization algorithm. In this case the caller traverses the DAG. If the graph is static, the caller can send all tasks with the DAG to the callee to reduce synchronization cost.

3.6 User-defined Filters

Complementary to a scheduler-based flow control, the parameter server supports user-defined filters to selectively synchronize individual (key,value) pairs, allowing fine-grained control of data consistency within a task. The insight is that the optimization algorithm itself usually possesses information on which parameters are most useful for synchronization. One example is the *significantly modified* filter, which only pushes entries that have changed by more than a threshold since their last synchronization. In Section 5.1, we discuss another filter named *KKT* which takes advantage of the optimality condition of the optimization problem: a worker only pushes gradients that are likely to affect the weights on the servers.

Algorithm 2 Set vector clock to t for range $\mathcal{R}$ and node i

1: **for** $\mathcal{S} \in \{\mathcal{S}_i : \mathcal{S}_i \cap \mathcal{R} \neq \emptyset, i = 1, \ldots, n\}$ **do**
2:     **if** $\mathcal{S} \subseteq \mathcal{R}$ **then** $\mathrm{vc}_i(\mathcal{S}) \leftarrow t$ **else**
3:         $a \leftarrow \max(\mathcal{S}^b, \mathcal{R}^b)$ and $b \leftarrow \min(\mathcal{S}^e, \mathcal{R}^e)$
4:         split range $\mathcal{S}$ into $[\mathcal{S}^b, a), [a, b), [b, \mathcal{S}^e)$
5:         $\mathrm{vc}_i([a, b)) \leftarrow t$
6:     **end if**
7: **end for**

4 Implementation

The servers store the parameters (key-value pairs) using consistent hashing [45] (Sec. 4.3). For fault tolerance, entries are replicated using chain replication [47] (Sec. 4.4). Different from prior (key,value) systems, the parameter server is optimized for *range based communication* with compression on both data (Sec. 4.2) and range based vector clocks (Sec. 4.1).

4.1 Vector Clock

Given the potentially complex task dependency graph and the need for fast recovery, each (key,value) pair is associated with a vector clock [30, 15], which records the time of each individual node on this (key,value) pair. Vector clocks are convenient, e.g., for tracking aggregation status or rejecting doubly sent data. However, a naive implementation of the vector clock requires $O(nm)$ space to handle n nodes and m parameters. With thousands of nodes and billions of parameters, this is infeasible in terms of memory and bandwidth.

Fortunately, many parameters hare the same timestamp as a result of the range-based communication pattern of the parameter server: If a node pushes the parameters in a range, then the timestamps of the parameters associated with the node are likely the same. Therefore, they can be compressed into a single range vector clock. More specifically, assume that $\mathrm{vc}_i(k)$ is the time of key k for node i. Given a key range $\mathcal{R}$, the ranged vector clock $\mathrm{vc}_i(\mathcal{R}) = t$ means for any key $k \in \mathcal{R}$, $\mathrm{vc}_i(k) = t$.

Initially, there is only one range vector clock for each node i. It covers the entire parameter key space as its

range with 0 as its initial timestamp. Each range set may split the range and create at most 3 new vector clocks (see Algorithm 2). Let k be the total number of unique ranges communicated by the algorithm, then there are at most $\mathcal{O}(mk)$ vector clocks, where m is the number of nodes. k is typically much smaller than the total number of parameters. This significantly reduces the space required for range vector clocks.[3]

4.2 Messages

Nodes may send messages to individual nodes or node groups. A message consists of a list of (key,value) pairs in the key range $\mathcal{R}$ and the associated range vector clock:

$$[\mathrm{vc}(\mathcal{R}), (k_1, v_1), \ldots, (k_p, v_p)] \;\; k_j \in \mathcal{R} \text{ and } j \in \{1, \ldots p\}$$

This is the basic communication format of the parameter server not only for shared parameters but also for tasks. For the latter, a (key,value) pair might assume the form (task ID, arguments or return results).

Messages may carry a subset of all available keys within range $\mathcal{R}$. The missing keys are assigned the same timestamp without changing their values. A message can be split by the key range. This happens when a worker sends a message to the whole server group, or when the key assignment of the receiver node has changed. By doing so, we partition the (key,value) lists and split the range vector clock similar to Algorithm 2.

Because machine learning problems typically require high bandwidth, message compression is desirable. Training data often remains unchanged between iterations. A worker might send the same key lists again. Hence it is desirable for the receiving node to cache the key lists. Later, the sender only needs to send a hash of the list rather than the list itself. Values, in turn, may contain many zero entries. For example, a large portion of parameters remain unchanged in sparse logistic regression, as evaluated in Section 5.1. Likewise, a user-defined filter may also zero out a large fraction of the values (see Figure 12). Hence we need only send nonzero (key,value) pairs. We use the fast Snappy compression library [21] to compress messages, effectively removing the zeros. Note that key-caching and value-compression can be used jointly.

4.3 Consistent Hashing

The parameter server partitions keys much as a conventional distributed hash table does [8, 41]: keys and server node IDs are both inserted into the hash ring (Figure 7). Each server node manages the key range starting with its insertion point to the next point by other nodes in the counter-clockwise direction. This node is called the master of this key range. A physical server is often represented in the ring via multiple "virtual" servers to improve load balancing and recovery.

We simplify the management by using a direct-mapped DHT design. The server manager handles the ring management. All other nodes cache the key partition locally. This way they can determine directly which server is responsible for a key range, and are notified of any changes.

4.4 Replication and Consistency

Each server node stores a replica of the k counterclockwise neighbor key ranges relative to the one it owns. We refer to nodes holding copies as slaves of the appropriate key range. The above diagram shows an example with $k = 2$, where server 1 replicates the key ranges owned by server 2 and server 3.

Worker nodes communicate with the master of a key range for both `push` and `pull`. Any modification on the master is copied with its timestamp to the slaves. Modifications to data are pushed synchronously to the slaves. Figure 8 shows a case where worker 1 pushes x into server 1, which invokes a user defined function f to modify the shared data. The push task is completed only once the data modification $f(x)$ is copied to the slave.

Naive replication potentially increases the network traffic by k times. This is undesirable for many machine learning applications that depend on high network bandwidth. The parameter server framework permits an important optimization for many algorithms: replication after aggregation. Server nodes often aggregate data from the worker nodes, such as summing local gradients. Servers may therefore postpone replication until aggregation is complete. In the righthand side of the diagram, two workers push x and y to the server, respectively. The server first aggregates the push by $x + y$, then applies the modification $f(x+y)$, and finally performs the replication. With n workers, replication uses only k/n bandwidth. Often k is a small constant, while n is hundreds to thousands. While aggregation increases the delay of the task reply, it can be hidden by relaxed consistency conditions.

4.5 Server Management

To achieve fault tolerance and dynamic scaling we must support addition and removal of nodes. For convenience we refer to virtual servers below. The following steps happen when a server joins.

[3] Ranges can be also merged to reduce the number of fragments. However, in practice both m and k are small enough to be easily handled. We leave merging for future work.

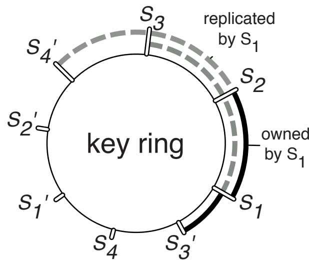


Figure 7: Server node layout.

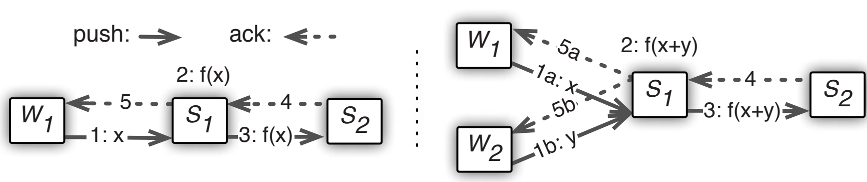


Figure 8: Replica generation. Left: single worker. Right: multiple workers updating values simultaneously.

1. The server manager assigns the new node a key range to serve as master. This may cause another key range to split or be removed from a terminated node.
2. The node fetches the range of data to maintains as master and k additional ranges to keep as slave.
3. The server manager broadcasts the node changes. The recipients of the message may shrink their own data based on key ranges they no longer hold and to resubmit unfinished tasks to the new node.

Fetching the data in the range $\mathcal{R}$ from some node S proceeds in two stages, similar to the Ouroboros protocol [38]. First S pre-copies all (key,value) pairs in the range together with the associated vector clocks. This may cause a range vector clock to split similar to Algorithm 2. If the new node fails at this stage, S remains unchanged. At the second stage S no longer accepts messages affecting the key range $\mathcal{R}$ by dropping the messages without executing and replying. At the same time, S sends the new node all changes that occurred in $\mathcal{R}$ during the pre-copy stage.

On receiving the node change message a node N first checks if it also maintains the key range $\mathcal{R}$. If true and if this key range is no longer to be maintained by N, it deletes all associated (key,value) pairs and vector clocks in $\mathcal{R}$. Next, N scans all outgoing messages that have not received replies yet. If a key range intersects with $\mathcal{R}$, then the message will be split and resent.

Due to delays, failures, and lost acknowledgements N may send messages twice. Due to the use of vector clocks both the original recipient and the new node are able to reject this message and it does not affect correctness.

The departure of a server node (voluntary or due to failure) is similar to a join. The server manager tasks a new node with taking the key range of the leaving node. The server manager detects node failure by a heartbeat signal. Integration with a cluster resource manager such as Yarn [17] or Mesos [23] is left for future work.

4.6 Worker Management

Adding a new worker node W is similar but simpler than adding a new server node:

1. The task scheduler assigns W a range of data.
2. This node loads the range of training data from a network file system or existing workers. Training data is often read-only, so there is no two-phase fetch. Next, W pulls the shared parameters from servers.
3. The task scheduler broadcasts the change, possibly causing other workers to free some training data.

When a worker departs, the task scheduler may start a replacement. We give the algorithm designer the option to control recovery for two reasons: If the training data is huge, recovering a worker node be may more expensive than recovering a server node. Second, losing a small amount of training data during optimization typically affects the model only a little. Hence the algorithm designer may prefer to continue without replacing a failed worker. It may even be desirable to terminate the slowest workers.

5 Evaluation

We evaluate our parameter server based on the use cases of Section 2 — Sparse Logistic Regression and Latent Dirichlet Allocation. We also show results of sketching to illustrate the generality of our framework. The experiments were run on clusters in two (different) large internet companies and a university research cluster to demonstrate the versatility of our approach.

5.1 Sparse Logistic Regression

Problem and Data: Sparse logistic regression is one of the most popular algorithms for large scale risk minimization [9]. It combines the logistic loss[4] with the ℓ_1

[4] $\ell(x_i, y_i, w) = \log(1 + \exp(-y_i \langle x_i, w \rangle))$

Algorithm 3 Delayed Block Proximal Gradient [31]

Scheduler:

1: Partition features into b ranges $\mathcal{R}_1, \ldots, \mathcal{R}_b$
2: **for** $t = 0$ **to** T **do**
3: Pick random range $\mathcal{R}_{i_t}$ and issue task to workers
4: **end for**

Worker r at iteration t

1: Wait until all iterations before $t - \tau$ are finished
2: Compute first-order gradient $g_r^{(t)}$ and diagonal second-order gradient $u_r^{(t)}$ on range $\mathcal{R}_{i_t}$
3: Push $g_r^{(t)}$ and $u_r^{(t)}$ to servers with the KKT filter
4: Pull $w_r^{(t+1)}$ from servers

Servers at iteration t

1: Aggregate gradients to obtain $g^{(t)}$ and $u^{(t)}$
2: Solve the proximal operator

$$w^{(t+1)} \leftarrow \underset{u}{\operatorname{argmin}}\, \Omega(u) + \frac{1}{2\eta}\|w^{(t)} - \eta g^{(t)} + u\|_H^2,$$

where $H = \operatorname{diag}(h^{(t)})$ and $\|x\|_H^2 = x^T H x$

	Method	Consistency	LOC
System A	L-BFGS	Sequential	10,000
System B	Block PG	Sequential	30,000
Parameter Server	Block PG	Bounded Delay KKT Filter	300

Table 3: Systems evaluated.

regularizer[5] of Section 2.2. The latter biases a compact solution with a large portion of 0 value entries. The non-smoothness of this regularizer, however, makes learning more difficult.

We collected an ad click prediction dataset with 170 billion examples and 65 billion unique features. This dataset is 636 TB uncompressed (141 TB compressed). We ran the parameter server on 1000 machines, each with 16 physical cores, 192GB DRAM, and connected by 10 Gb Ethernet. 800 machines acted as workers, and 200 were parameter servers. The cluster was in concurrent use by other (unrelated) tasks during operation.

Algorithm: We used a state-of-the-art distributed regression algorithm (Algorithm 3, [31, 32]). It differs from the simpler variant described earlier in four ways: First, only a block of parameters is updated in an iteration. Second, the workers compute both gradients and the diagonal part of the second derivative on this block. Third, the parameter servers themselves must perform complex computation: the servers update the model by solving a *proximal operator* based on the aggregated local gradients. Fourth, we use a bounded-delay model over iterations and use a "KKT" filter to suppress transmission of parts of the generated gradient update that are small enough that their effect is likely to be negligible.[6]

[5] $\Omega(w) = \sum_{i=1}^n |w_i|$

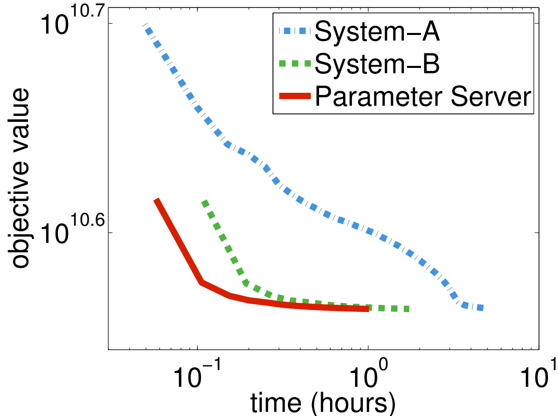


Figure 9: Convergence of sparse logistic regression. The goal is to minimize the objective rapidly.

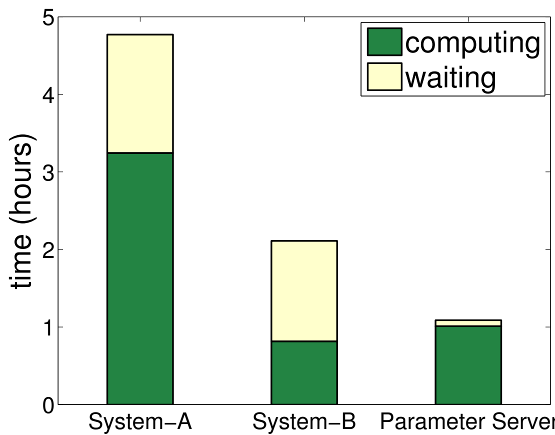


Figure 10: Time per worker spent on computation and waiting during sparse logistic regression.

To the best of our knowledge, no open source system can scale sparse logistic regression to the scale described in this paper.[7] We compare the parameter server with two special-purpose systems, named System A and B, devel-

[6] A user-defined Karush-Kuhn-Tucker (KKT) filter [26]. Feature k is filtered if $w_k = 0$ and $|\hat{g}_k| \leq \Delta$. Here $\hat{g}_k$ is an estimate of the global gradient based on the worker's local information and $\Delta > 0$ is a user-defined parameter.

[7] Graphlab provides only a multi-threaded, single machine implementation, while Petuum, Mlbase and REEF do not support sparse logistic regression. We confirmed this with the authors as per 4/2014.

oped by a large internet company.

Notably, both Systems A and B consist of more than 10K lines of code. The parameter server only requires 300 lines of code for the same functionality as System B.[8] The parameter server successfully moves most of the system complexity from the algorithmic implementation into a reusable generalized component.

Results: We first compare these three systems by running them to reach the same objective value. A better system achieves a lower objective in less time. Figure 9 shows the results: System B outperforms system A because it uses a better algorithm. The parameter server, in turn, outperforms System B while using the same algorithm. It does so because of the efficacy of reducing the network traffic and the relaxed consistency model.

Figure 10 shows that the relaxed consistency model substantially increases worker node utilization. Workers can begin processing the next block without waiting for the previous one to finish, hiding the delay otherwise imposed by barrier synchronization. Workers in System A are 32% idle, and in system B, they are 53% idle, while waiting for the barrier in each block. The parameter server reduces this cost to under 2%. This is not entirely free: the parameter server uses slightly more CPU than System B for two reasons. First, and less fundamentally, System B optimizes its gradient calculations by careful data preprocessing. Second, asynchronous updates with the parameter server require more iterations to achieve the same objective value. Due to the significantly reduced communication cost, the parameter server halves the total time.

Next we evaluate the reduction of network traffic by each system components. Figure 11 shows the results for servers and workers. As can be seen, allowing the senders and receivers to cache the keys can save near 50% traffic. This is because both key (`int64`) and value (`double`) are of the same size, and the key set is not changed during optimization. In addition, data compression is effective for compressing the values for both servers (>20x) and workers when applying the KKT filter (>6x). The reason is twofold. First, the ℓ_1 regularizer encourages a sparse model (w), so that most of values pulled from servers are 0. Second, the KKT filter forces a large portion of gradients sending to servers to be 0. This can be seen more clearly in Figure 12, which shows that more than 93% unique features are filtered by the KKT filter.

Finally, we analyze the bounded delay consistency model. The time decomposition of workers to achieve the same convergence criteria under different maximum allowed delay (τ) is shown in Figure 13. As expected, the waiting time decreases when the allowed delay increases. Workers are 50% idle when using the sequential consistency model ($\tau = 0$), while the idle rate is reduced to 1.7% when τ is set to be 16. However, the computing time increases nearly linearly with τ. Because the data inconsistency slows convergence, more iterations are needed to achieve the same convergence criteria. As a result, $\tau = 8$ is the best trade-off between algorithm convergence and system performance.

[8] System B was developed by an author of this paper.

5.2 Latent Dirichlet Allocation

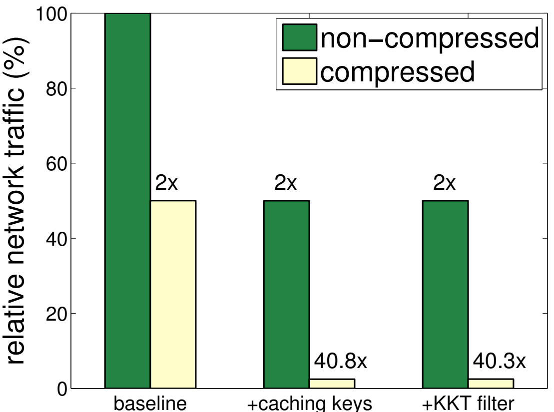
Problem and Data: To demonstrate the versatility of our approach, we applied the same parameter server architecture to the problem of modeling user interests based upon which domains appear in the URLs they click on in search results. We collected search log data containing 5 billion unique user identifiers and evaluated the model for the 5 million most frequently clicked domains in the result set. We ran the algorithm using 800 workers and 200 servers and 5000 workers and 1000 servers respectively. The machines had 10 physical cores, 128GB DRAM, and at least 10 Gb/s of network connectivity. We again shared the cluster with production jobs running concurrently.

Algorithm: We performed LDA using a combination of Stochastic Variational Methods [25], Collapsed Gibbs sampling [20] and distributed gradient descent. Here, gradients are aggregated asynchronously as they arrive from workers, along the lines of [1].

We divided the parameters in the model into local and global parameters. The local parameters (i.e. auxiliary metadata) are pertinent to a given user and they are streamed the from disk whenever we access a given user. The global parameters are shared among users and they are represented as (key,value) pairs to be stored using the parameter server. User data is sharded over workers. Each of them runs a set of computation threads to perform inference over its assigned users. We synchronize asynchronously to send and receive local updates to the server and receive new values of the global parameters.

To our knowledge, no other system (e.g., YahooLDA, Graphlab or Petuum) can handle this amount of data and model complexity for LDA, using up to 10 billion (5 million tokens and 2000 topics) shared parameters. The largest previously reported experiments [2] had under 100 million users active at any time, less than 100,000 tokens and under 1000 topics (2% the data, 1% the parameters).

Results: To evaluate the quality of the inference algorithm we monitor how rapidly the training log-likelihood



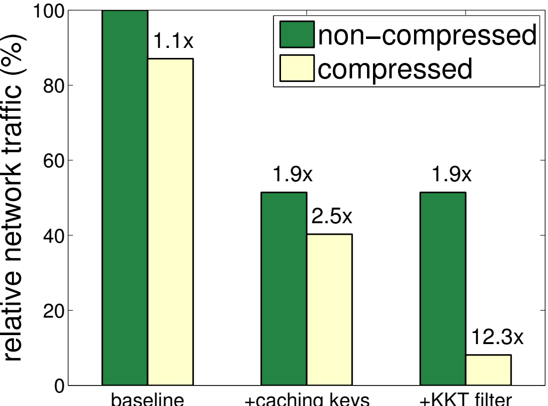


Figure 11: The savings of outgoing network traffic by different components. Left: per server. Right: per worker.

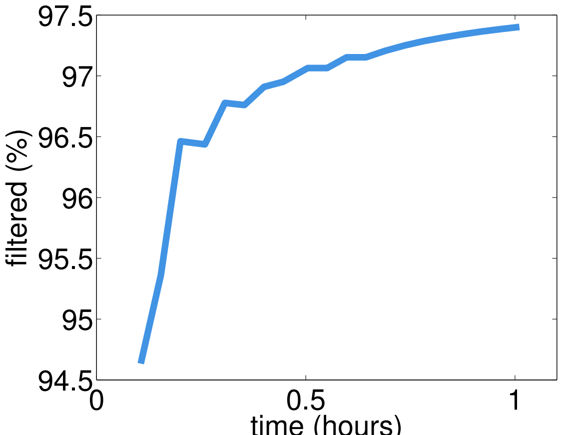


Figure 12: Unique features (keys) filtered by the KKT filter as optimization proceeds.

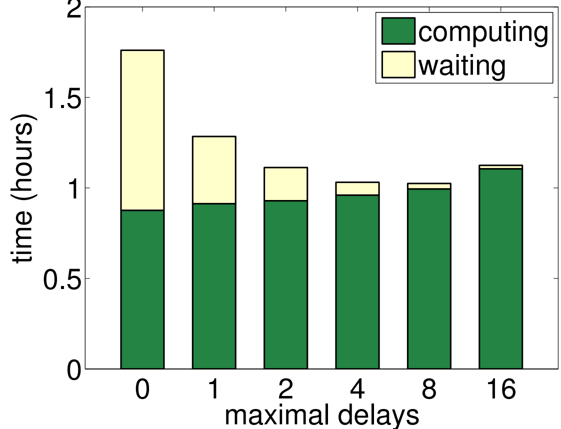


Figure 13: Time a worker spent to achieve the same convergence criteria by different maximal delays.

(measuring goodness of fit) converges. As can be seen in Figure 14, we observe an approximately 4x speedup in convergence when increasing the number of machines from 1000 to 6000. The stragglers observed in Figure 14 (leftmost) also illustrate the importance of having an architecture that can cope with performance variation across workers.

Topic name	# Top urls
Programming	stackoverflow.com w3schools.com cplusplus.com github.com tutorialspoint.com jquery.com codeproject.com oracle.com qt-project.org bytes.com android.com mysql.com
Music	ultimate-guitar.com guitaretab.com 911tabs.com e-chords.com songsterr.com chordify.net musicnotes.com ukulele-tabs.com
Baby Related	babycenter.com whattoexpect.com babycentre.co.uk circleofmoms.com thebump.com parents.com momtastic.com parenting.com americanpregnancy.org kidshealth.org
Strength Training	bodybuilding.com muscleandfitness.com mensfitness.com menshealth.com t-nation.com livestrong.com muscleandstrength.com myfitnesspal.com elitefitness.com crossfit.com steroid.com gnc.com askmen.com

Table 4: Example topics learned using LDA over the .5 billion dataset. Each topic represents a user interest

5.3 Sketches

Problem and Data: We include sketches as part of our evaluation as a test of generality, because they operate very differently from machine learning algorithms. They typically observe a large number of writes of events coming from a streaming data source [11, 5].

We evaluate the time required to insert a streaming log of pageviews into an approximate structure that can efficiently track pageview counts for a large collection of web pages. We use the Wikipedia (and other Wiki projects) page view statistics as benchmark. Each entry is an unique key of a webpage with the corresponding number of requests served in a hour. From 12/2007 to 1/2014, there are 300 billion entries for more than 100 million unique keys. We run the parameter server with 90 virtual server nodes on 15 machines of a research cluster [40] (each has

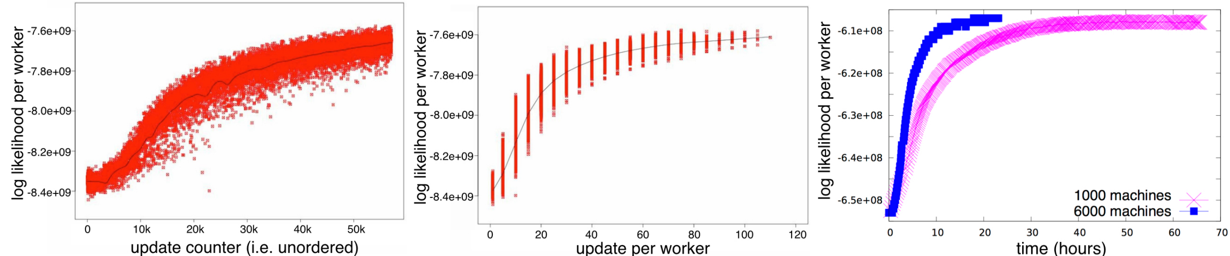


Figure 14: Left: Distribution over worker log-likelihoods as a function of time for 1000 machines and 5 billion users. Some of the low values are due to stragglers synchronizing slowly initially. Middle: the same distribution, stratified by the number of iterations. Right: convergence (time in 1000s) using 1000 and 6000 machines on 500M users.

Algorithm 4 CountMin Sketch

Init: $M[i, j] = 0$ **for** $i \in \{1, \ldots n\}$ **and** $j \in \{1, \ldots k\}$.

Insert(x)

1: **for** $i = 1$ **to** k **do**

2: $\quad M[i, \text{hash}(i, x)] \leftarrow M[i, \text{hash}(i, x)] + 1$

Query(x)

1: **return** $\min \{M[i, \text{hash}(i, x)] \text{ for } 1 \leq i \leq k\}$

64 cores and is connected by a 40Gb Ethernet).

Algorithm: Sketching algorithms efficiently store summaries of huge volumes of data so that approximate queries can be quickly answered. These algorithms are particularly important in streaming applications where data and queries arrive in real-time. Some of the highest-volume applications involve examples such as Cloudflare's DDoS-prevention service, which must analyze page requests across its entire content delivery service architecture to identify likely DDoS targets and attackers. The volume of data logged in such applications considerably exceeds the capacity of a single machine. While a conventional approach might be to shard a workload across a key-value cluster such as Redis, these systems typically do not allow the user-defined aggregation semantics needed to implement *approximate* aggregation.

Algorithm 4 gives a brief overview of the CountMin sketch [11]. By design, the result of a query is an *upper* bound on the number of observed keys x. Splitting keys into ranges automatically allows us to parallelize the sketch. Unlike the two previous applications, the workers simply dispatch updates to the appropriate servers.

Results: The system achieves very high insert rates, which are shown in Table 5. It performs well for two reasons: First, bulk communication reduces the communication cost. Second, message compression reduces the average (key,value) size to around 50 bits. Importantly, when we terminated a server node during the insertion, the parameter server was able to recover the failed node within 1 second, making our system well equipped for realtime.

Metric	Value
Peak inserts per second	1.3 billion
Average inserts per second	1.1 billion
Peak net bandwidth per machine	4.37 GBit/s
Time to recover a failed node	0.8 second

Table 5: Results of distributed CountMin

6 Summary and Discussion

We described a parameter server framework to solve distributed machine learning problems. This framework is easy to use: Globally shared parameters can be used as local sparse vectors or matrices to perform linear algebra operations with local training data. It is efficient: All communication is asynchronous. Flexible consistentcy models are supported to balance the trade-off between system efficiency and fast algorithm convergence rate. Furthermore, it provides elastic scalability and fault tolerance, aiming for stable long term deployment. Finally, we show experiments for several challenging tasks on real datasets with billions of variables to demonstrate its efficiency. We believe that this third generation parameter server is an important building block for scalable machine learning. The codes are available at `parameterserver.org`.

Acknowledgments: This work was supported in part by gifts and/or machine time from Google, Amazon, Baidu, PRObE, and Microsoft; by NSF award 1409802; and by the Intel Science and Technology Center for Cloud Computing. We are grateful to our reviewers and colleagues for their comments on earlier versions of this paper.

References

[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of The 5th ACM International Conference on Web Search and Data Mining (WSDM)*, 2012.

[2] A. Ahmed, Y. Low, M. Aly, V. Josifovski, and A. J. Smola. Scalable inference of dynamic user interests for behavioural targeting. In *Knowledge Discovery and Data Mining*, 2011.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.

[4] Apache Foundation. Mahout project, 2012. http://mahout.apache.org.

[5] R. Berinde, G. Cormode, P. Indyk, and M.J. Strauss. Space-optimal heavy hitters with strong error bounds. In J. Paredaens and J. Su, editors, *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*, pages 157–166. ACM, 2009.

[6] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[7] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, January 2003.

[8] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Peer-to-peer systems II*, pages 80–87. Springer, 2003.

[9] K. Canini. Sibyl: A system for large scale supervised machine learning. *Technical Talk*, 2012.

[10] B.-G. Chun, T. Condie, C. Curino, C. Douglas, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, J. Rosen, R. Sears, and M. Weimer. Reef: Retainable evaluator execution framework. *Proceedings of the VLDB Endowment*, 6(12):1370–1373, 2013.

[11] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SDM*, 2005.

[12] W. Dai, J. Wei, X. Zheng, J. K. Kim, S. Lee, J. Yin, Q. Ho, and E. P. Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013.

[13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems*, 2012.

[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In T. C. Bressoud and M. F. Kaashoek, editors, *Symposium on Operating Systems Principles*, pages 205–220. ACM, 2007.

[16] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[17] The Apache Software Foundation. Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/.

[18] The Apache Software Foundation. Apache hadoop, 2009. http://hadoop.apache.org/core/.

[19] F. Girosi, M. Jones, and T. Poggio. Priors, stabilizers and basis functions: From regularization to radial, tensor and additive splines. A.I. Memo 1430, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1993.

[20] T.L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004.

[21] S. H. Gunderson. Snappy: A fast compressor/decompressor. https://code.google.com/p/snappy/.

[22] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 2 edition, 2009.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22, 2011.

[24] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[25] M. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. In *International Conference on Machine Learning*, 2012.

[26] W. Karush. Minima of functions of several variables with inequalities as side constraints. Master's thesis, Dept. of Mathematics, Univ. of Chicago, 1939.

[27] L. Kim. How many ads does Google serve in a day?, 2012. http://goo.gl/oIidXO.

[28] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[29] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[30] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[31] M. Li, D. G. Andersen, and A. J. Smola. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013.

[32] M. Li, D. G. Andersen, and A. J. Smola. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Neural Information Processing Systems*, 2014.

[33] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D.G. Andersen, and A. J. Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, 2013.

[34] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed Graphlab: A framework for machine learning and data mining in the cloud. In *PVLDB*, 2012.

[35] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, and D. Golovin. Ad click prediction: a view from the trenches. In *KDD*, 2013.

[36] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT Press, 2012.

[37] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[38] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini. Flex-KV: Enabling high-performance and flexible KV systems. In *Proceedings of the 2012 workshop on Management of big data systems*, pages 19–24. ACM, 2012.

[39] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In R. H. Arpaci-Dusseau and B. Chen, editors, *Operating Systems Design and Implementation, OSDI*, pages 293–306. USENIX Association, 2010.

[40] PRObE Project. Parallel Reconfigurable Observational Environment. https://www.nmc-probe.org/wiki/Machines:Susitna,

[41] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[42] B. Schölkopf and A. J. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

[43] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010.

[44] E. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. Mli: An api for distributed machine learning. 2013.

[45] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[46] C.H. Teo, Q. Le, A. J. Smola, and S. V. N. Vishwanathan. A scalable modular convex solver for regularized risk minimization. In *Proc. ACM Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 2007.

[47] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[48] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[49] R.C. Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[50] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. M. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Fast and interactive analytics over Hadoop data with Spark. *USENIX ;login:*, 37(4):45–51, August 2012.

GraphX: Graph Processing in a Distributed Dataflow Framework

Joseph E. Gonzalez*, Reynold S. Xin*†, Ankur Dave*, Daniel Crankshaw*
Michael J. Franklin*, Ion Stoica*†
*UC Berkeley AMPLab †Databricks

Abstract

In pursuit of graph processing performance, the systems community has largely abandoned general-purpose distributed dataflow frameworks in favor of specialized graph processing systems that provide tailored programming abstractions and accelerate the execution of iterative graph algorithms. In this paper we argue that many of the advantages of specialized graph processing systems can be recovered in a modern general-purpose distributed dataflow system. We introduce GraphX, an *embedded* graph processing framework built on top of Apache Spark, a widely used distributed dataflow system. GraphX presents a familiar composable graph abstraction that is sufficient to express existing graph APIs, yet can be implemented using only a few basic dataflow operators (e.g., join, map, group-by). To achieve performance parity with specialized graph systems, GraphX recasts graph-specific optimizations as distributed join optimizations and materialized view maintenance. By leveraging advances in distributed dataflow frameworks, GraphX brings low-cost fault tolerance to graph processing. We evaluate GraphX on real workloads and demonstrate that GraphX achieves an order of magnitude performance gain over the base dataflow framework and matches the performance of specialized graph processing systems while enabling a wider range of computation.

1 Introduction

The growing scale and importance of graph data has driven the development of numerous specialized *graph processing* systems including Pregel [22], PowerGraph [13], and many others [7, 9, 37]. By exposing specialized abstractions backed by graph-specific optimizations, these systems can naturally express and efficiently execute iterative graph algorithms like PageRank [30] and community detection [18] on graphs with billions of vertices and edges. As a consequence, graph processing systems typically outperform general-purpose distributed dataflow frameworks like Hadoop MapReduce by orders of magnitude [13, 20].

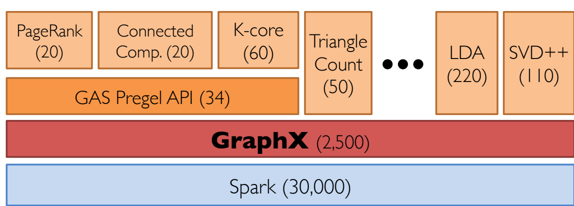


Figure 1: **GraphX** is a thin layer on top of the Spark general-purpose dataflow framework (lines of code).

While the restricted focus of these systems enables a wide range of system optimizations, it also comes at a cost. Graphs are only part of the larger analytics process which often combines graphs with unstructured and tabular data. Consequently, analytics pipelines (e.g., Figure 11) are forced to compose multiple systems which increases complexity and leads to unnecessary data movement and duplication. Furthermore, in pursuit of performance, graph processing systems often abandon fault tolerance in favor of snapshot recovery. Finally, as specialized systems, graph processing frameworks do not generally enjoy the broad support of distributed dataflow frameworks.

In contrast, general-purpose distributed dataflow frameworks (e.g., Map-Reduce [10], Spark [39], Dryad [15]) expose rich dataflow operators (e.g., map, reduce, group-by, join), are well suited for analyzing unstructured and tabular data, and are widely adopted. However, *directly* implementing iterative graph algorithms using dataflow operators can be challenging, often requiring multiple stages of complex joins. Furthermore, the *general-purpose* join and aggregation strategies defined in distributed dataflow frameworks do not leverage the common patterns and structure in iterative graph algorithms and therefore miss important optimization opportunities.

Historically, graph processing systems evolved separately from distributed dataflow frameworks for several reasons. First, the early emphasis on single stage computation and on-disk processing in distributed dataflow frameworks (e.g., MapReduce) limited their applicability to iterative graph algorithms which repeatedly and randomly access subsets of the graph. Second, early distributed dataflow frameworks did not expose fine-grained control over the data partitioning, hindering the application of graph partitioning techniques. However, new in-memory distributed dataflow frameworks (e.g., Spark and Naiad) expose control over data partitioning and in-memory representation, addressing some of these limitations.

Given these developments, we believe there is an opportunity to unify advances in graph processing systems with advances in dataflow systems enabling a single system to address the entire analytics pipeline. In this paper we explore the design of graph processing systems on top of general purpose distributed dataflow systems. We argue that by identifying the essential dataflow patterns in graph computation and recasting optimizations in graph processing systems as dataflow optimizations we can recover the advantages of specialized graph processing systems within a general-purpose distributed dataflow framework. To support this argument we introduce GraphX, an efficient graph processing framework embedded within the Spark [39] distributed dataflow system.

GraphX presents a familiar, expressive graph API (Section 3). Using the GraphX API we implement a variant of the popular Pregel abstraction as well as a range of common graph operations. Unlike existing graph processing systems, the GraphX API enables the *composition* of graphs with unstructured and tabular data and permits the same physical data to be viewed both as a graph and as collections without data movement or duplication. For example, using GraphX it is easy to join a social graph with user comments, apply graph algorithms, and expose the results as either collections or graphs to other procedures (e.g., visualization or rollup). Consequently, GraphX enables users to adopt the computational pattern (graph or collection) that is best suited for the current task without sacrificing performance or flexibility.

We built GraphX as a library on top of Spark (Figure 1) by encoding graphs as collections and then expressing the GraphX API on top of standard dataflow operators. GraphX requires no modifications to Spark, revealing a general method to embed graph computation within distributed dataflow frameworks and distill graph computation to a specific *join–map–group-by* dataflow pattern. By reducing graph computation to a specific pattern we identify the critical path for system optimization.

However, naively encoding graphs as collections and executing iterative graph computation using general-purpose dataflow operators can be slow and inefficient. To achieve performance parity with specialized graph processing systems, GraphX introduces a range of optimizations (Section 4) both in how graphs are encoded as collections as well as the execution of the common dataflow operators. Flexible vertex-cut partitioning is used to encode graphs as horizontally partitioned collections and match the state of the art in distributed graph partitioning. GraphX recasts system optimizations developed in the context of graph processing systems as join optimizations (e.g., CSR indexing, join elimination, and join-site specification) and materialized view maintenance (e.g., vertex mirroring and delta updates) and applies these techniques to the Spark dataflow operators. By leveraging logical partitioning and lineage, GraphX achieves low-cost fault tolerance. Finally, by exploiting immutability GraphX reuses indices across graph and collection views and over multiple iterations, reducing memory overhead and improving system performance.

We evaluate GraphX on real-world graphs and compare against direct implementations of graph algorithms using the Spark dataflow operators as well as implementations using specialized graph processing systems. We demonstrate that GraphX can achieve performance parity with specialized graph processing systems while preserving the advantages of a general-purpose dataflow framework. In summary, the contributions of this paper are:

1. an integrated graph and collections API which is sufficient to express existing graph abstractions and enable a much wider range of computation.
2. an embedding of vertex-cut partitioned graphs in horizontally partitioned collections and the GraphX API in a small set of general-purpose dataflow operators.
3. distributed join and materialized view optimizations that enable general-purpose distributed dataflow frameworks to execute graph computation at performance parity with specialized graph systems.
4. a large-scale evaluation on real graphs and common benchmarking algorithms comparing GraphX against widely used graph processing systems.

2 Background

In this section we review the design trade-offs and limitations of graph processing systems and distributed dataflow frameworks. At a high level, graph processing systems define computation at the granularity of vertices and their neighborhoods and exploit the sparse dependency structure pre-defined by the graph. In contrast, general-purpose distributed dataflow frameworks define computation as dataflow operators at either the granularity of individual items (e.g., filter, map) or across entire collections (i.e., operations like non-broadcast join that require a shuffle).

2.1 The Property Graph Data Model

Graph processing systems represent graph structured data as a **property graph** [33], which associates user-defined properties with each vertex and edge. The properties can include meta-data (e.g., user profiles and time stamps) and program state (e.g., the PageRank of vertices or inferred affinities). Property graphs derived from natural phenomena such as social networks and web graphs often have highly skewed, power-law degree distributions and orders of magnitude more edges than vertices [18].

In contrast to dataflow systems whose operators (e.g., join) can span multiple collections, operations in graph processing systems (e.g., vertex programs) are typically defined with respect to a *single* property graph with a *pre-declared, sparse structure*. While this restricted focus facilitates a range of optimizations (Section 2.3), it also complicates the expression of analytics tasks that may span multiple graphs and sub-graphs.

2.2 The Graph-Parallel Abstraction

Algorithms ranging from PageRank to latent factor analysis iteratively transform vertex properties based on the properties of adjacent vertices and edges. This common pattern of *iterative local transformations* forms the basis of the graph-parallel abstraction. In the graph-parallel abstraction [13], a user-defined **vertex program** is instantiated concurrently for each vertex and interacts with adjacent vertex programs through messages (e.g., Pregel [22]) or shared state (e.g., PowerGraph [13]). Each vertex program can read and modify its vertex property and in some cases [13, 20] adjacent vertex properties. When all vertex programs vote to halt the program terminates.

As a concrete example, in Listing 1 we express the PageRank algorithm as a Pregel vertex program. The vertex program for the vertex v begins by summing the messages encoding the weighted PageRank of neighboring vertices. The PageRank is updated using the resulting sum and is then broadcast to its neighbors (weighted by the number of links). Finally, the vertex program assesses whether it has converged (locally) and votes to halt.

The extent to which vertex programs run concurrently differs across systems. Most systems (e.g., [7, 13, 22, 34]) adopt the bulk synchronous execution model, in which all vertex programs run concurrently in a sequence of super-steps. Some systems (e.g., [13, 20, 37]) also support an asynchronous execution model that mitigates the effect of stragglers by running vertex programs as resources become available. However, the gains due to an asynchronous programming model are often offset by the additional complexity and so we focus on the bulk-synchronous model and rely on system level techniques (e.g., pipelining and speculation) to address stragglers.

```
def PageRank(v: Id, msgs: List[Double]) {
  // Compute the message sum
  var msgSum = 0
  for (m <- msgs) { msgSum += m }
  // Update the PageRank
  PR(v) = 0.15 + 0.85 * msgSum
  // Broadcast messages with new PR
  for (j <- OutNbrs(v)) {
    msg = PR(v) / NumLinks(v)
    send_msg(to=j, msg)
  }
  // Check for termination
  if (converged(PR(v))) voteToHalt(v)
}
```

Listing 1: **PageRank in Pregel**: computes the sum of the inbound messages, updates the PageRank value for the vertex, and then sends the new weighted PageRank value to neighboring vertices. Finally, if the PageRank did not change the vertex program votes to halt.

While the graph-parallel abstraction is well suited for iterative graph algorithms that respect the static neighborhood structure of the graph (e.g., PageRank), it is not well suited to express computation where disconnected vertices interact or where computation changes the graph structure. For example, tasks such as graph construction from raw text or unstructured data, graph coarsening, and analysis that spans multiple graphs are difficult to express in the vertex centric programming model.

2.3 Graph System Optimizations

The restrictions imposed by the graph-parallel abstraction along with the sparse graph structure enable a range of important system optimizations.

The GAS Decomposition: Gonzalez et al. [13] observed that most vertex programs interact with neighboring vertices by collecting messages in the form of a generalized commutative associative sum and then broadcasting new messages in an inherently parallel loop. They proposed the GAS decomposition which splits vertex programs into three *data-parallel* stages: Gather, Apply, and Scatter. In Listing 2 we decompose the PageRank vertex program into the Gather, Apply, and Scatter stages.

The GAS decomposition leads to a *pull-based* model of message computation: the system asks the vertex program for value of the message between *adjacent vertices* rather than the user sending messages directly from the vertex program. As a consequence, the GAS decomposition enables vertex-cut partitioning, improved work balance, serial edge-iteration [34], and reduced data movement. However, the GAS decomposition also prohibits direct communication between vertices that are not adjacent in the graph and therefore hinders the expression of more general communication patterns.

```
def Gather(a: Double, b: Double) = a + b
def Apply(v, msgSum) {
  PR(v) = 0.15 + 0.85 * msgSum
  if (converged(PR(v))) voteToHalt(v)
}
def Scatter(v, j) = PR(v) / NumLinks(v)
```

Listing 2: **Gather-Apply-Scatter (GAS) PageRank**: The gather phase combines inbound messages. The apply phase consumes the final message sum and updates the vertex property. The scatter phase defines the message computation for each edge.

Graph Partitioning: Graph processing systems apply a range of graph-partitioning algorithms [16] to minimize communication and balance computation. Gonzalez et al. [13] demonstrated that vertex-cut [12] partitioning performs well on many large natural graphs. Vertex-cut partitioning evenly assigns edges to machines in a way that minimizes the number of times each vertex is cut.

Mirror Vertices: Often high-degree vertices will have multiple neighbors on the same remote machine. Rather than sending multiple, typically identical, messages across the network, graph processing systems [13, 20, 24, 32] adopt mirroring techniques in which a single message is sent to the mirror and then forwarded to all the neighbors. Graph processing systems exploit the static graph structure to reuse the mirror data structures.

Active Vertices: As graph algorithms proceed, vertex programs within a graph converge at different rates, leading to rapidly shrinking working sets (the collection of active vertex programs). Recent systems [11, 13, 20, 22] track active vertices and eliminate data movement and unnecessary computation for vertices that have converged. In addition, these systems typically maintain efficient densely packed data-structures (e.g., compressed sparse row (CSR)) that enable constant-time access to the local edges adjacent to active vertices.

2.4 Distributed Dataflow Frameworks

We use the term distributed dataflow framework to refer to cluster compute frameworks like MapReduce and its generalizations. Although details vary from one framework to another, they typically satisfy the following properties:

1. a data model consisting of typed collections (i.e., a generalization of tables to unstructured data).

2. a coarse-grained data-parallel programming model composed of deterministic operators which transform collections (e.g., map, group-by, and join).

3. a scheduler that breaks each job into a directed acyclic graph (DAG) of tasks, where each task runs on a (horizontal) partition of data.

4. a runtime that can tolerate stragglers and partial cluster failures without restarting.

In MapReduce, the programming model exposes only two dataflow operators: *map* and *reduce* (a.k.a., group-by). Each job can contain at most two layers in its DAG of tasks. More modern frameworks such as DryadLINQ [15], Pig [29], and Spark expose additional dataflow operators such as fold and join, and can execute tasks with multiple layers of dependencies.

Distributed dataflow frameworks have enjoyed broad adoption for a wide variety of data processing tasks, including ETL, SQL query processing, and iterative machine learning. They have also been shown to scale to thousands of nodes operating on petabytes of data.

In this work we restrict our attention to Apache Spark, upon which we developed GraphX. Spark has several features that are particularly attractive for GraphX:

1. The Spark storage abstraction called Resilient Distributed Datasets (RDDs) enables applications to keep data in memory, which is essential for iterative graph algorithms.

2. RDDs permit user-defined data partitioning, and the execution engine can exploit this to co-partition RDDs and co-schedule tasks to avoid data movement. This is essential for encoding partitioned graphs.

3. Spark logs the lineage of operations used to build an RDD, enabling automatic reconstruction of lost partitions upon failures. Since the lineage graph is relatively small even for long-running applications, this approach incurs negligible runtime overhead, unlike checkpointing, and can be left on without concern for performance. Furthermore, Spark supports optional in-memory distributed replication to reduce the amount of recomputation on failure.

4. Spark provides a high-level API in Scala that can be easily extended. This aided in creating a coherent API for both collections and graphs.

We believe that many of the ideas in GraphX could be applied to other contemporary dataflow systems and in Section 6 we discuss some preliminary work on a GraphLINQ, a graph framework within Naiad.

3 The GraphX Programming Abstraction

We now revisit graph computation from the perspective of a general-purpose dataflow framework. We recast the property graph data model as collections and the graph-parallel abstraction as a specific pattern of dataflow operators. In the process we reveal the essential structure of graph-parallel computation and identify the key operators required to execute graph algorithms efficiently.

3.1 Property Graphs as Collections

The property graph, described in Section 2.1, can be logically represented as a pair of vertex and edge property collections. The *vertex collection* contains the vertex properties uniquely keyed by the vertex identifier. In the GraphX system, vertex identifiers are 64-bit integers which may be derived externally (e.g., user ids) or by applying a hash function to the vertex property (e.g., page URL). The *edge collection* contains the edge properties keyed by the source and destination vertex identifiers.

By reducing the property graph to a pair of collections we make it possible to compose graphs with other collections in a distributed dataflow framework. Operations like adding additional vertex properties are naturally expressed as joins against the vertex property collection. The process of analyzing the results of graph computation (i.e., the final vertex and edge properties) and comparing properties across graphs becomes as simple as analyzing and joining the corresponding collections. Both of these tasks are routine in the broader scope of graph analytics but are not well served by the graph parallel abstraction.

New property graphs can be constructed by composing different vertex and edge property collections. For example, we can construct logically distinct graphs with separate vertex properties (e.g., one storing PageRanks and another storing connected component membership) while sharing the same edge collection. This may appear to be a small accomplishment, but the tight integration of vertices and edges in specialized graph processing systems often hinders even this basic form of reuse. In addition, graph-specific index data structures can be shared across graphs with common vertex and edge collections, reducing storage overhead and improving performance.

3.2 Graph Computation as Dataflow Ops.

The normalized representation of a property graph as a pair of vertex and edge property collections allows us to embed graphs in a distributed dataflow framework. In this section we describe how dataflow operators can be composed to express graph computation.

Graph-parallel computation, introduced in Section 2.2, is the process of computing aggregate properties of the neighborhood of each vertex (e.g., the sum of the PageRanks of neighboring vertices weighted by the edge values). We can express graph-parallel computation in a distributed dataflow framework as a sequence of *join stages* and *group-by stages* punctuated by map operations.

In the *join stage*, vertex and edge properties are joined to form the *triplets view*[1] consisting of each edge and its corresponding source and destination vertex properties.

[1] The *triplet* terminology derives from the classic Resource Description Framework (RDF), discussed in Section 6.

```
CREATE VIEW triplets AS
SELECT s.Id, d.Id, s.P, e.P, d.P
FROM edges AS e
JOIN vertices AS s JOIN vertices AS d
ON e.srcId = s.Id AND e.dstId = d.Id
```

Listing 3: **Constructing Triplets in SQL**: The column `P` represents the properties in the vertex and edge property collections.

The triplets view is best illustrated by the SQL statement in Listing 3, which constructs the triplets view as a three way join keyed by the source and destination vertex ids.

In the *group-by stage*, the triplets are grouped by source or destination vertex to construct the *neighborhood* of each vertex and compute aggregates. For example, to compute the PageRank of a vertex we would execute:

```
SELECT t.dstId, 0.15+0.85*sum(t.srcP*t.eP)
FROM triplets AS t GROUP BY t.dstId
```

By iteratively applying the above query to update the vertex properties until they converge, we can calculate the PageRank of each vertex.

These two stages capture the GAS decomposition described in Section 2.3. The group-by stage *gathers* messages destined to the same vertex, an intervening map operation *applies* the message sum to update the vertex property, and the join stage *scatters* the new vertex property to all adjacent vertices.

Similarly, we can implement the GAS decomposition of the Pregel abstraction by iteratively composing the join and group-by stages with data-parallel map stages. Each iteration begins by executing the join stage to bind active vertices with their outbound edges. Using the triplets view, messages are computed along each triplet in a map stage and then aggregated at their destination vertex in a group-by stage. Finally, the messages are received by the vertex programs in a map stage over the vertices.

The dataflow embedding of the Pregel abstraction demonstrates that graph-parallel computation can be expressed in terms of a simple sequence of *join* and *group-by* dataflow operators. Additionally, it stresses the need to efficiently maintain the triplets view in the join stage and compute the neighborhood aggregates in the group-by stage. Consequently, these stages are the focus of performance optimization in graph processing systems. We describe how to implement them efficiently in Section 4.

3.3 GraphX Operators

The GraphX programming abstraction extends the Spark dataflow operators by introducing a small set of specialized *graph operators*, summarized in Listing 4.

```
class Graph[V, E] {
  // Constructor
  def Graph(v: Collection[(Id, V)],
            e: Collection[(Id, Id, E)])
  // Collection views
  def vertices: Collection[(Id, V)]
  def edges: Collection[(Id, Id, E)]
  def triplets: Collection[Triplet]
  // Graph-parallel computation
  def mrTriplets(f: (Triplet) => M,
      sum: (M, M) => M): Collection[(Id, M)]
  // Convenience functions
  def mapV(f: (Id, V) => V): Graph[V, E]
  def mapE(f: (Id, Id, E) => E): Graph[V, E]
  def leftJoinV(v: Collection[(Id, V)],
      f: (Id, V, V) => V): Graph[V, E]
  def leftJoinE(e: Collection[(Id, Id, E)],
      f: (Id, Id, E, E) => E): Graph[V, E]
  def subgraph(vPred: (Id, V) => Boolean,
      ePred: (Triplet) => Boolean)
    : Graph[V, E]
  def reverse: Graph[V, E]
}
```

Listing 4: **Graph Operators:** transform vertex and edge collections.

The `Graph` constructor logically binds together a pair of vertex and edge property collections into a property graph. It also verifies the integrity constraints: that every vertex occurs only once and that edges do not link missing vertices. Conversely, the `vertices` and `edges` operators expose the graph's vertex and edge property collections. The `triplets` operator returns the triplets view (Listing 3) of the graph as described in Section 3.2. If a triplets view already exists, the previous triplets are incrementally maintained to avoid a full join (see Section 4.2).

The `mrTriplets` (Map Reduce Triplets) operator encodes the essential two-stage process of graph-parallel computation defined in Section 3.2. Logically, the `mrTriplets` operator is the composition of the *map* and *group-by* dataflow operators on the triplets view. The user-defined map function is applied to each triplet, yielding a value (i.e., a message of type `M`) which is then aggregated at the destination vertex using the user-defined binary aggregation function as illustrated in the following:

```
SELECT t.dstId, reduceF(mapF(t)) AS msgSum
FROM triplets AS t GROUP BY t.dstId
```

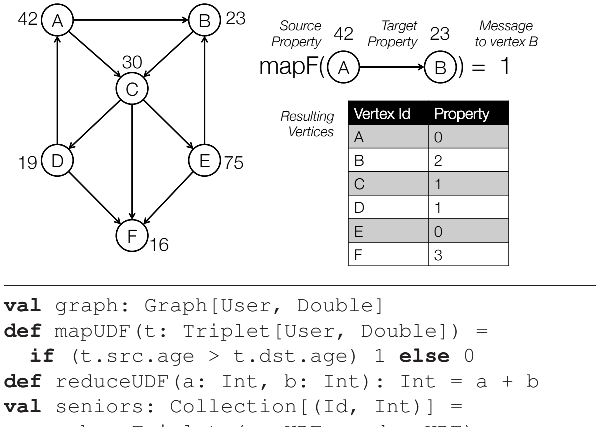
The `mrTriplets` operator produces a collection containing the sum of the inbound messages keyed by the destination vertex identifier. For example, in Figure 2 we use the `mrTriplets` operator to compute a collection containing the number of older followers for each user in a social network. Because the resulting collection contains a subset of the vertices in the graph it can reuse the same indices as the original vertex collection.

Finally, Listing 4 contains several functions that simply perform a dataflow operation on the vertex or edge collections. We define these functions only for caller convenience; they are not essential to the abstraction and can easily be defined using standard dataflow operators. For example, `mapV` is defined as follows:

```
g.mapV(f) ≡ Graph(g.vertices.map(f), g.edges)
```

In Listing 5 we use the GraphX API to implement a GAS decomposition of the Pregel abstraction. We begin by initializing the vertex properties with an additional field to track active vertices (those that have not voted to halt). Then, while there are active vertices, messages are computed using the `mrTriplets` operator and the vertex program is applied to the resulting message sums.

By expressing message computation as an *edge-parallel* map operation followed by a commutative associative aggregation, we leverage the GAS decomposition




```
val graph: Graph[User, Double]
def mapUDF(t: Triplet[User, Double]) =
  if (t.src.age > t.dst.age) 1 else 0
def reduceUDF(a: Int, b: Int): Int = a + b
val seniors: Collection[(Id, Int)] =
  graph.mrTriplets(mapUDF, reduceUDF)
```

Figure 2: **Example use of mrTriplets**: Compute the number of older followers of each vertex.

```
def Pregel(g: Graph[V, E],
      vprog: (Id, V, M) => V,
      sendMsg: (Triplet) => M,
      gather: (M, M) => M): Collection[V] = {
 // Set all vertices as active
 g = g.mapV((id, v) => (v, halt=false))
 // Loop until convergence
 while (g.vertices.exists(v => !v.halt)) {
   // Compute the messages
   val msgs: Collection[(Id, M)] =
     // Restrict to edges with active source
     g.subgraph(ePred=(s,d,sP,eP,dP)=>!sP.halt)
     // Compute messages
      .mrTriplets(sendMsg, gather)
   // Receive messages and run vertex program
   g = g.leftJoinV(msgs).mapV(vprog)
 }
 return g.vertices
}
```

Listing 5: **GraphX Enhanced Pregel:** An implementation of the Pregel abstraction using the GraphX API.

```
def ConnectedComp(g: Graph[V, E]) = {
  g = g.mapV(v => v.id) // Initialize vertices
  def vProg(v: Id, m: Id): Id = {
    if (v == m) voteToHalt(v)
    return min(v, m)
  }
  def sendMsg(t: Triplet): Id =
    if (t.src.cc < t.dst.cc) t.src.cc
    else None // No message required
  def gatherMsg(a: Id, b: Id): Id = min(a, b)
  return Pregel(g, vProg, sendMsg, gatherMsg)
}
```

Listing 6: **Connected Components:** For each vertex we compute the lowest reachable vertex id using Pregel.

to mitigate the cost of high-degree vertices. Furthermore, by exposing the entire triplet to the message computation we can simplify algorithms like connected components. However, in cases where the entire triplet is not needed (e.g., PageRank which requires only the source property) we rely on UDF bytecode inspection (see Section 4.3.2) to automatically drop unused fields from join.

In Listing 6 we use the GraphX variant of Pregel to implement the connected components algorithm. The connected components algorithm computes the lowest reachable vertex id for each vertex. We initialize the vertex property of each vertex to equal its id using `mapV` and then define the three functions required to use the GraphX Pregel API. The `sendMsg` function leverages the triplet view of the edge to only send a message to neighboring vertices when their component id should change. The `gatherMsg` function computes the minimum of the inbound message values and the vertex program (`vProg`) determines the new component id.

Combining Graph and Collection Operators: Often groups of connected vertices are better modeled as a single vertex. In these cases, it can be helpful coarsen the graph by aggregating connected vertices that share a common characteristic (e.g., web domain) to derive a new graph (e.g., the domain graph). We use the GraphX abstraction to implement graph coarsening in Listing 7.

The coarsening operation takes an edge predicate and a vertex aggregation function and collapses all edges that satisfy the predicate, merging their respective vertices. The edge predicate is used to first construct the subgraph of edges that are to be collapsed (i.e., modifying the graph structure). Then the graph-parallel connected components algorithm is run on the subgraph. Each connected component corresponds to a super-vertex in the new coarsened graph with the component id being the lowest vertex id in the component. The super-vertices are constructed by aggregating all the vertices with the same component id using a data-parallel aggregation operator. Finally, we update the edges to link together super-vertices and generate the new graph for subsequent graph-parallel computation.

```
def coarsen(g: Graph[V, E],
            pred: (Id,Id,V,E,V) => Boolean,
            reduce: (V,V) => V) = {
  // Restrict graph to contractable edges
  val subG = g.subgraph(v => True, pred)
  // Compute connected component id for all V
  val cc: Collection[(Id,ccId)] =
    ConnectedComp(subG).vertices
  // Merge all vertices in same component
  val superV: Collection[(ccId,V)] =
    g.vertices.leftJoin(cc)
              .groupBy(CC_ID, reduce)
  // Link remaining edges between components
  val invG = g.subgraph(ePred = t => !pred(t))
  val remainingE: Collection[(ccId,ccId,E)] =
    invG.leftJoin(cc).triplets.map {
      e => (e.src.cc, e.dst.cc, e.attr)
    }
  // Return the final graph
  Graph(superV, remainingE)
}
```

Listing 7: **Coarsen:** The *coarsening* operator merges vertices connected by edges that satisfy the edge predicate.

The *coarsen* operator demonstrates the power of a unified abstraction by combining both data-parallel and graph-parallel operators in a single graph-analytics task.

4 The GraphX System

GraphX achieves performance parity with specialized graph processing systems by recasting the graph-specific optimizations of Section 2.3 as optimizations on top of a small set of standard dataflow operators in Spark. In this section we describe these optimizations in the context of classic techniques in traditional database systems including indexing, incremental view maintenance, and join optimizations. Along the way, we quantify the effectiveness of each optimization; readers are referred to Section 5 for details on datasets and experimental setup.

4.1 Distributed Graph Representation

GraphX represents graphs internally as a pair of vertex and edge collections built on the Spark RDD abstraction. These collections introduce indexing and graph-specific partitioning as a layer on top of RDDs. Figure 3 illustrates the physical representation of the horizontally partitioned vertex and edge collections and their indices.

The **vertex collection** is hash-partitioned by the vertex ids. To support frequent joins across vertex collections, vertices are stored in a local hash index within each partition (Section 4.2). Additionally, a bitmask stores the visibility of each vertex, enabling soft deletions to promote index reuse (Section 4.3.1).

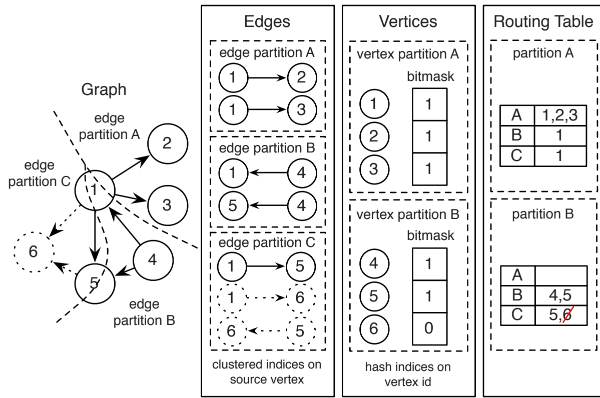


Figure 3: **Distributed Graph Representation**: The graph (left) is represented as a vertex and an edge collection (right). The edges are divided into three edge partitions by applying a partition function (e.g., 2D Partitioning). The vertices are partitioned by vertex id. Co-partitioned with the vertices, GraphX maintains a routing table encoding the edge partitions for each vertex. If vertex 6 and adjacent edges (shown with dotted lines) are restricted from the graph (e.g., by `subgraph`), they are removed from the corresponding collection by updating the bitmasks thereby enabling index reuse.

The **edge collection** is horizontally partitioned by a user-defined partition function. GraphX enables *vertex-cut* partitioning, which minimizes communication in natural graphs such as social networks and web graphs [13]. By default edges are assigned to partitions based on the partitioning of the input collection (e.g., the original placement on HDFS). However, GraphX provides a range of built-in partitioning functions, including a 2D hash partitioner with strong upper bounds [8] on the communication complexity of operators like `mrTriplets`. This flexibility in edge placement is enabled by the *routing table*, described in Section 4.2. For efficient lookup of edges by their source and target vertices, the edges within a partition are clustered by source vertex id using a *compressed sparse row* (CSR) [35] representation and hash-indexed by their target id. Section 4.3.1 discusses how these indices accelerate iterative computation.

Index Reuse: GraphX inherits the immutability of Spark and therefore all graph operators *logically* create new collections rather than destructively modifying existing ones. As a result, derived vertex and edge collections can often share indices to reduce memory overhead and accelerate local graph operations. For example, the hash index on vertices enables fast aggregations, and the resulting aggregates share the index with the original vertices.

In addition to reducing memory overhead, shared indices enable faster joins. Vertex collections sharing the same index (e.g., the vertices and the messages from `mrTriplets`) can be joined by a coordinated scan, similar to a merge join, without requiring any index lookups. In our benchmarks, index reuse reduces the per-iteration runtime of PageRank on the Twitter graph by 59%.

The GraphX operators try to maximize index reuse. Operators that do not modify the graph structure (e.g., `mapV`) automatically preserve indices. To reuse indices for operations that *restrict* the graph structure (e.g., `subgraph`), GraphX relies on bitmasks to construct restricted views. In cases where index reuse could lead to decreased efficiency (e.g., when a graph is highly filtered), GraphX uses the *reindex* operator to build new indices.

4.2 Implementing the Triplets View

As described in Section 3.2, a key stage in graph computation is constructing and maintaining the *triplets view*, which consists of a three-way join between the source and destination vertex properties and the edge properties.

Vertex Mirroring: Because the vertex and edge property collections are partitioned independently, the join requires data movement. GraphX performs the three-way join by shipping the vertex properties across the network to the edges, thus setting the edge partitions as the *join sites* [21]. This approach substantially reduces communication for two reasons. First, real-world graphs commonly have orders of magnitude more edges than vertices. Second, a single vertex may have many edges in the same partition, enabling substantial reuse of the vertex property.

Multicast Join: While *broadcast join* in which all vertices are sent to each edge partition would ensure joins occur on edge partitions, it could still be inefficient since most partitions require only a small subset of the vertices to complete the join. Therefore, GraphX introduces a *multicast join* in which each vertex property is sent only to the edge partitions that contain adjacent edges. For each vertex GraphX maintains the set of edge partitions with adjacent edges. This join site information is stored in a *routing table* which is co-partitioned with the vertex collection (Figure 3). The routing table is associated with the edge collection and constructed lazily upon first instantiation of the triplets view.

The flexibility in partitioning afforded by the multicast join strategy enables more sophisticated application-specific graph partitioning techniques. For example, by adopting a per-city partitioning scheme on the Facebook social network graph Ugander et al. [38] showed a 50.5% reduction in query time. In Section 5.1 we exploit the optimized partitioning of our sample datasets to achieve up to 56% reduction in runtime and 5.8× reduction in communication compared to a 2D hash partitioning.

Partial Materialization: Vertex replication is performed eagerly when vertex properties change, but the local joins at the edge partitions are left unmaterialized to

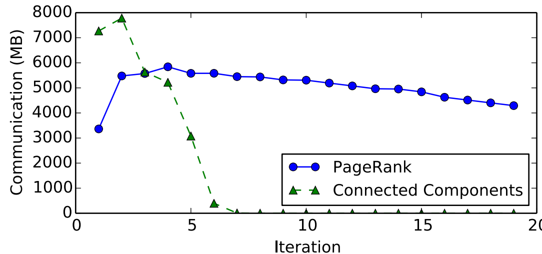


Figure 4: **Impact of incrementally maintaining the triplets view**: For both PageRank and connected components, as vertices converge, communication decreases due to incremental view maintenance. The initial rise in communication is due to message compression (Section 4.4); many PageRank values are initially the same.

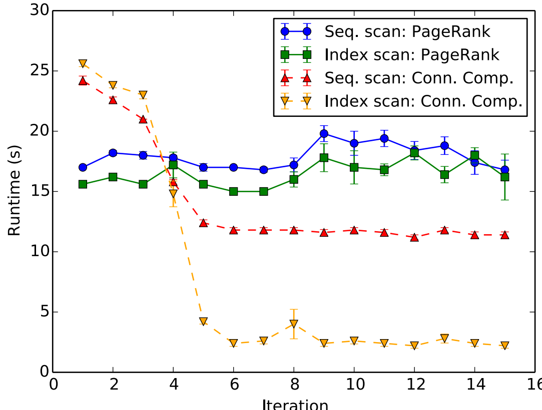


Figure 5: **Sequential scan vs index scan**: Connected components on the Twitter graph benefits greatly from switching to index scan after the 4th iteration, while PageRank benefits only slightly because the set of active vertices is large even at the 15th iteration.

avoid duplication. Instead, mirrored vertex properties are stored in hash maps on each edge partition and referenced when constructing triplets.

Incremental View Maintenance: Iterative graph algorithms often modify only a subset of the vertex properties in each iteration. We therefore apply *incremental view maintenance* to the triplets view to avoid unnecessary movement of unchanged data. After each graph operation, we track which vertex properties have changed since the triplets view was last constructed. When the triplets view is next accessed, only the changed vertices are re-routed to their edge-partition join sites and the local mirrored values of the unchanged vertices are reused. This functionality is managed automatically by the graph operators.

Figure 4 illustrates the impact of incremental view maintenance for both PageRank and connected components on the Twitter graph. In the case of PageRank, where the number of active vertices decreases slowly because the convergence threshold was set to 0, we see only moderate gains. In contrast, for connected components most vertices are within a short distance of each other and converge quickly, leading to a substantial reduction in communication from incremental view maintenance. Without incremental view maintenance, the triplets view would need to be reconstructed from scratch every iteration, and communication would remain at its peak throughout the computation.

4.3 Optimizations to mrTriplets

GraphX incorporates two additional query optimizations for the `mrTriplets` operator: *filtered index scanning* and *automatic join elimination*.

4.3.1 Filtered Index Scanning

The first stage of the `mrTriplets` operator logically involves a scan of the triplets view to apply the user-defined map function to each triplet. However, as iterative graph algorithms converge, their working sets tend to shrink, and the map function skips all but a few triplets. In particular, the map function only needs to operate on triplets containing vertices in the *active set*, which is defined by an application-specific predicate. Directly scanning all triplets becomes increasingly wasteful as the active set shrinks. For example, in the last iteration of connected components on the Twitter graph, only a few of the vertices are still active. However, to execute `mrTriplets` we still must sequentially scan 1.5 billion edges and check whether their vertices are in the active set.

To address this problem, we introduced an indexed scan for the triplets view. The application expresses the current active set by restricting the graph using the `subgraph` operator. The vertex predicate is pushed to the edge partitions, where it can be used to filter the triplets using the CSR index on the source vertex id (Section 4.1). We measure the selectivity of the vertex predicate and switch from sequential scan to clustered index scan when the selectivity is less than 0.8.

Figure 5 illustrates the benefit of index scans in PageRank and connected components. As with incremental view maintenance, index scans lead to a smaller improvement in runtime for PageRank and a substantial improvement in runtime for connected components. Interestingly, in the initial iterations of connected components, when the majority of the vertices are active, a sequential scan is slightly faster as it does not require the additional index lookup. It is for this reason that we dynamically switch between full and indexed scans based on the fraction of active vertices.

4.3.2 Automatic Join Elimination

In some cases, operations on the triplets view may access only one of the vertex properties or none at all. For example, when `mrTriplets` is used to count the degree of each vertex, the map UDF does not access any vertex properties. Similarly, when computing messages in PageRank only the source vertex properties are used.

GraphX uses a JVM bytecode analyzer to inspect user-defined functions at runtime and determine whether the source or target vertex properties are referenced. If only one property is referenced, and if the triplets view has not already been materialized, GraphX automatically rewrites the query plan for generating the triplets view from a three-way join to a two-way join. If none of the vertex properties are referenced, GraphX eliminates the join entirely. This modification is possible because the triplets view follows the lazy semantics of RDDs in Spark. If the user never accesses the triplets view, it is never materialized. A call to `mrTriplets` is therefore able to rewrite the join needed to generate the relevant part of the triplets view.

Figure 6 demonstrates the impact of this physical execution plan rewrite on communication and runtime for PageRank on the Twitter follower graph. We see that join elimination cuts the amount of data transferred in half, leading to a significant reduction in overall runtime. Note that on the first iteration there is no reduction in communication. This is due to compression algorithms that take advantage of all messages having exactly the same initial value. However, compression and decompression still consume CPU time so we still observe nearly a factor of two reduction in overall runtime.

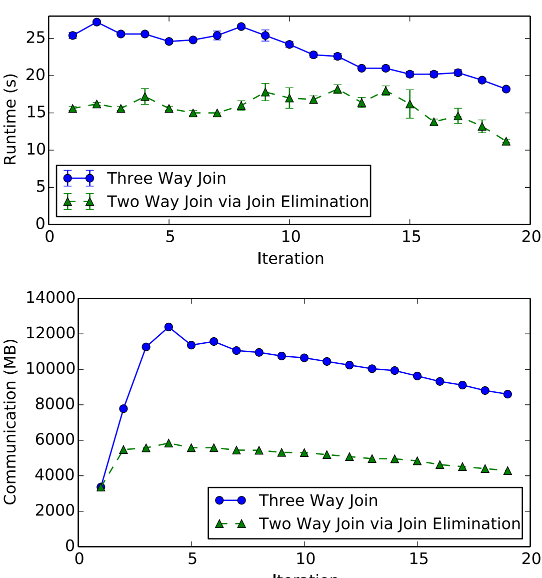


Figure 6: **Impact of automatic join elimination on communication and runtime**: We ran PageRank for 20 iterations on the Twitter dataset with and without join elimination and found that join elimination reduces the amount of communication by almost half and substantially decreases the total execution time.

4.4 Additional Optimizations

While implementing GraphX, we discovered that a number of low level engineering details had significant performance impact. We sketch some of them here.

Memory-based Shuffle: Spark's default shuffle implementation materializes the temporary data to disk. We modified the shuffle phase to materialize map outputs in memory and remove this temporary data using a timeout.

Batching and Columnar Structure: In our join code path, rather than shuffling the vertices one by one, we batch a block of vertices routed to the same target join site and convert the block from row-oriented format to column-oriented format. We then apply the LZF compression algorithm on these blocks to send them. Batching has a negligible impact on CPU time while improving the compression ratio of LZF by 10–40% in our benchmarks.

Variable Integer Encoding: While GraphX uses 64-bit vertex ids, in most cases the ids are much smaller than 2^{64}. To exploit this fact, during shuffling, we encode integers using a variable-encoding scheme where for each byte, we use only the first 7 bits to encode the value, and use the highest order bit to indicate whether we need another byte to encode the value. In this case, smaller integers are encoded with fewer bytes. In the worst case, integers greater than 2^{56} require 5 bytes to encode. This technique reduces communication in PageRank by 20%.

5 System Evaluation

In this section we demonstrate that, for iterative graph algorithms, GraphX is over an order of magnitude faster than directly using the general-purpose dataflow operators described in Section 3.2 and is comparable to or faster than specialized graph processing systems.

We evaluate the performance of GraphX on several graph-analytics tasks, comparing it with the following:

1. **Apache Spark 0.9.1**: the base distributed dataflow system for GraphX. We compare against Spark to demonstrate the performance gains relative to the baseline distributed dataflow framework.
2. **Apache Giraph 1.1**: an open source graph computation system based on the Pregel abstraction.
3. **GraphLab 2.2 (PowerGraph)**: the open-source graph computation system based on the GAS decomposition of vertex programs. Because GraphLab

is implemented in C++ and all other systems run on the JVM, given identical optimizations, we would expect GraphLab to have a slight performance advantage.

We also compare against GraphLab without shared-memory parallelism (denoted **GraphLab NoSHM**). GraphLab communicates between workers on the same machine using shared data structures. In contrast, Giraph, Spark, and GraphX adopt a shared-nothing worker model incurring extra serialization overhead between workers. To isolate this overhead, we disabled shared-memory by forcing GraphLab workers to run in separate processes.

It is worth noting that the shared data structures in GraphLab increase the complexity of the system. Indeed, we encountered and fixed a critical bug in one of the GraphLab shared data structures. The resulting patch introduced an additional lock which led to a small increase in thread contention. As a consequence, in some cases (e.g., Figure 7c) disabling shared memory contributed to a small improvement in performance.

All experiments were conducted on Amazon EC2 using 16 `m2.4xlarge` worker nodes. Each node has 8 virtual cores, 68 GB of memory, and two hard disks. The cluster was running 64-bit Linux 3.2.28. We plot the mean and standard deviation for multiple trials of each experiment.

5.1 System Comparison

Cross-system benchmarks are often unfair due to the difficulty in tuning each system equitably. We have endeavored to minimize this effect by working closely with experts in each of the systems to achieve optimal configurations. We emphasize that we are *not* claiming GraphX is fundamentally faster than GraphLab or Giraph; these systems could in theory implement the same optimizations as GraphX. Instead, we aim to show that it is possible to achieve comparable performance to specialized graph processing systems using a general dataflow engine while gaining common dataflow features such as fault tolerance.

While we have implemented a wide range of graph algorithms on top of GraphX, we restrict our performance evaluation to PageRank and connected components. These two representative graph algorithms are implemented in most graph processing systems, have well-understood behavior, and are simple enough to serve as an effective measure of the system's performance. To ensure a fair comparison, our PageRank implementation is based on Listing 1; it does not exploit delta messages and therefore benefits less from indexed scans and incremental view maintenance. Conversely, the connected components implementation only sends messages when a vertex must change component membership and therefore does benefit from incremental view maintenance.

Dataset	Edges	Vertices
twitter-2010 [5, 4]	1,468,365,182	41,652,230
uk-2007-05 [5, 4]	3,738,733,648	105,896,555

Table 1: **Graph Datasets.** Both graphs have highly skewed power-law degree distributions.

For each system, we ran both algorithms on the twitter-2010 and uk-2007-05 graphs (Table 1). For Giraph and GraphLab we used the included implementations of these algorithms. For Spark we implemented the algorithms both using idiomatic dataflow operators (**Naive Spark**, as described in Section 3.2) and using an optimized implementation (**Optimized Spark**) that eliminates movement of edge data by pre-partitioning the edges to match the partitioning adopted by GraphX.

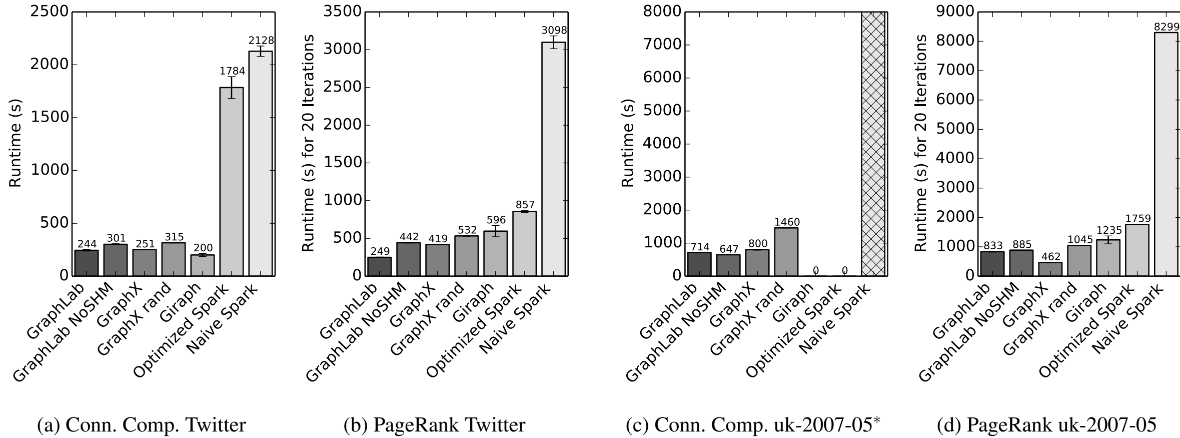
Both GraphLab and Giraph partition the graph according to specialized partitioning algorithms. While GraphX supports arbitrary user defined graph partitioners including those used by GraphLab and Giraph, the default partitioning strategy is to construct a vertex-cut that matches the input edge data layout thereby minimizing edge data movement when constructing the graph. However, as point of comparison we also tested GraphX using a randomized vertex-cut (**GraphX Rand**). We found (see Figure 7) that for the specific datasets used in our experiments the input partitioning, which was determined by a specialized graph compression format [4], actually resulted in a more communication-efficient vertex-cut partitioning.

Figures 7a and 7c show the total runtimes for connected components algorithm. We have excluded *Giraph* and *Optimized Spark* from Figure 7c because they were unable to scale to the larger web-graph in the allotted memory of the cluster. While the basic Spark implementation did not crash, it was forced to re-compute blocks from disk and exceeded 8000 seconds per iteration. We attribute the increased memory overhead to the use of edge-cut partitioning and the need to store bi-directed edges and messages for the connected components algorithm.

Figures 7b and 7d show the total runtimes for PageRank for 20 iterations on each system. In Figure 7b, GraphLab outperforms GraphX largely due to shared-memory parallelism; GraphLab without shared memory parallelism is much closer in performance to GraphX. In 7d, GraphX outperforms GraphLab because the input partitioning of uk-2007-05 is highly efficient, resulting in a 5.8x reduction in communication per iteration.

5.2 GraphX Performance

Scaling: In Figure 8 we evaluate the strong scaling performance of GraphX running PageRank on the Twitter follower graph. As we move from 8 to 32 machines (a factor of 4) we see a 3x speedup. However as we move to 64 machines (a factor of 8) we only see a 3.5x speedup.



(a) Conn. Comp. Twitter (b) PageRank Twitter (c) Conn. Comp. uk-2007-05* (d) PageRank uk-2007-05

Figure 7: **System Performance Comparison.** (c) Spark did not finish within 8000 seconds, Giraph and Spark + Part. ran out of memory.

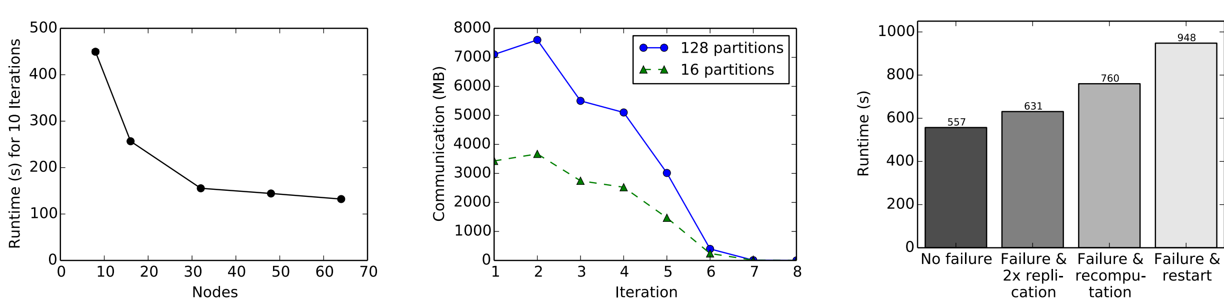


Figure 8: **Strong scaling for PageRank on Twitter (10 Iterations)**

Figure 9: **Effect of partitioning on communication**

Figure 10: **Fault tolerance for PageRank on uk-2007-05**

While this is hardly linear scaling, it is actually slightly better than the 3.2x speedup reported by GraphLab [13]. The poor scaling performance of PageRank has been attributed by [13] to high communication overhead relative to computation for the PageRank algorithm.

It may seem surprising that GraphX scales slightly better than GraphLab given that Spark does not exploit shared memory parallelism and therefore forces the graph to be partitioned across processors rather than machines. However, Figure 9 shows the communication of GraphX as a function of the number of partitions. Going from 16 to 128 partitions (a factor of 8) yields only an approximately 2-fold increase in communication. Returning to the analysis of vertex-cut partitioning conducted by [13], we find that the vertex-cut partitioning adopted by GraphX mitigates the 8-fold increase in communication.

Fault tolerance: Existing graph systems only support checkpoint-based fault tolerance, which most users leave disabled due to the performance overhead. GraphX is built on Spark, which provides lineage-based fault tolerance with negligible overhead as well as optional dataset replication (Section 2.4). We benchmarked these fault tolerance options for PageRank on uk-2007-05 by killing a worker in iteration 11 of 20, allowing Spark to recover by using the remaining copies of the lost partitions or recomputing them, and measuring how long the job took in total. For comparison, we also measured the end-to-end time for running until failure and then restarting from scratch on the remaining nodes using a driver script, as would be necessary in existing graph systems. Figure 10 shows that in case of failure, both replication and recomputation are faster than restarting the job from scratch, and moreover they are performed transparently by the dataflow engine.

6 Related Work

In Section 2 we described the general characteristics shared across many of the earlier graph processing systems. However, there are some exceptions to many of these characteristics that are worth noting.

While most of the work on large-scale distributed graph processing has focused on static graphs, several systems have focused on various forms of stream processing. One

of the earlier examples is Kineograph [9], a distributed graph processing system that constructs incremental snapshots of the graph for offline static graph analysis. In the multicore setting, GraphChi [17] and later X-Stream [34] introduced support for the addition of edges between existing vertices and between computation stages. Although conceptually GraphX could support the incremental introduction of edges (and potentially vertices), the existing data-structures would require additional optimization. Instead, GraphX focuses on efficiently supporting the *removal* of edges and vertices: essential functionality for offline sub-graph analysis.

Most of the optimizations and programming models of earlier graph processing systems focus on a single graph setting. While some of these systems [19, 13, 34] are capable of operating on multiple graphs *independently*, they do not expose an API or present optimizations for operations spanning graphs (or tables). One notable exception is CombBLAS [7] which treats graphs (and data more generally) as matrices and supports generalized binary algebraic operators. In contrast GraphX preserves the native semantics of graphs and tables and provides a simple API to combine data across these representations.

The triplets view in GraphX is related to the classic *Resource Description Framework* [23] (RDF) data model which encodes graph structured data as *subject-predicate-object* triplets (e.g., *NYC-isA-city*). Numerous systems [1, 6, 28] have been proposed for storing and executing SPARQL [31] subgraph queries against RDF triplets. Like GraphX, these systems rely heavily on indexing and clustering for performance. Unlike GraphX, these systems are not distributed or do not address iterative graph algorithms. Nonetheless, we believe that the optimizations techniques developed for GraphX may benefit the design of distributed graph query processing.

There have been several recent efforts at exploring graph algorithms within dataflow systems. Najork et al. [27], compares implementations of a range of graph algorithms on the DryadLINQ [15] and SQL Server dataflow systems. However, the resulting implementations are fairly complex and specialized, and little is discussed about graph-specific optimizations. Both Ewen et al. [11] and Murray et al. [26] proposed dataflow systems geared towards incremental iterative computation and demonstrated performance gains for specialized implementations of graph algorithms. While this work highlights the importance of incremental updates in graph computation, neither proposed a general method to express graph algorithms or graph specific optimizations beyond incremental dataflows. Nonetheless, we believe that the GraphX system could be ported to run on-top of these dataflow frameworks and would potentially benefit from advances like timely dataflows [26].

At the time of publication, the Microsoft Naiad team had announced initial work on a system called GraphLINQ [25], a graph processing framework on-top of Naiad which shares many similarities to GraphX. Like GraphX, GraphLINQ aims to provides rich graph functionality within a general-purpose dataflow framework. In particular GraphLINQ presents a `GraphReduce` operator that is semantically similar to the `mrTriplets` operator in GraphX except that it operates on streams of vertices and edges. The emphasis on stream processing exposes opportunities for classic optimizations in the stream processing literature as well as recent developments like the Naiad timely dataflows [26]. We believe this further supports the advantages of embedding graph processing within more general-purpose data processing systems.

Others have explored join optimizations in distributed dataflow frameworks. Blanas et al. [3] show that broadcast joins and semi-joins compare favorably with the standard MapReduce style shuffle joins when joining a large table (e.g., edges) with a smaller table (e.g., vertices). Closely related is the work by Afrati et al. [2] which explores optimizations for multi-way joins in a MapReduce framework. They consider joining a large relation with multiple smaller relations and provide a partitioning and replication strategy similar to classic 2D partitioning [8]. However, in contrast to our work, they do not construct a routing table forcing the system to broadcast the smaller relations (e.g., the vertices) to all partitions of the larger relation (e.g., the edges) that *could* have matching tuples. Furthermore, they force a particular hash partitioning on the larger relation precluding the opportunity for user defined graph partitioning algorithms (e.g., [16, 38, 36]).

7 Discussion

The work on GraphX addressed several key themes in data management systems and system design:

Physical Data Independence: GraphX allows the same physical data to be viewed as collections and as graphs without data movement or duplication. As a consequence the user is free to adopt the best view for the immediate task. We demonstrated that operations on collections and graphs can be efficiently implemented using the same physical representation and underlying operators. Our experiments show that this common substrate can match the performance of specialized graph systems.

Graph Computation as Joins and Group-By: The design of the GraphX system reveals the strong connection between distributed graph computation and distributed join optimizations. When viewed through the lens of dataflow operators, graph computation reduces to join and group-by operators. These two operators correspond to the Scatter and Gather stages of the GAS abstraction. Likewise, the optimizations developed for graph processing systems reduce to indexing, distributed join

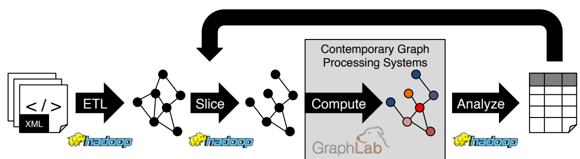


Figure 11: **Graph Analytics Pipeline:** requires multiple collection and graph views of the same data.

site selection, multicast joins, partial materialization, and incremental view maintenance.

The Narrow Waist: In designing the GraphX abstraction, we sought to develop a thin extension on top of dataflow operators with the goal of identifying the essential data model and core operations needed to support graph computation. We aimed for a portable framework that could be embedded in a range of dataflow frameworks. We believe that the GraphX design can be adopted by other dataflow systems, including MPP databases, to efficiently support a wide range of graph computations.

Analytics Pipelines: GraphX provides the ability to stay within a single framework throughout the analytics process, eliminating the need to learn and support multiple systems (e.g., Figure 11) or write data interchange formats and plumbing to move between systems. As a consequence, it is substantially easier to iteratively slice, transform, and compute on large graphs as well as to share data-structures across stages of the pipeline. The gains in performance and scalability for graph computation translate to a tighter analytics feedback loop and therefore a more efficient work flow.

Adoption: GraphX was publicly released as part of the 0.9.0 release of the Apache Spark open-source project.[2] It has since generated substantial interest in the community and has been used in production at various places.[3] Despite its nascent state, there has been considerable open-source contribution to GraphX with contributors providing some of the core graph functionality. We attribute this to its wide applicability and the simple abstraction built on top of an existing, popular dataflow framework.

8 Conclusions and Future Work

In this work we introduced GraphX, an efficient graph processing system that enables distributed dataflow frameworks such as Spark to naturally express and efficiently execute iterative graph algorithms. We identified a simple pattern of *join–map–group-by* dataflow operators that forms the basis of graph-parallel computation. Inspired by this observation, we proposed the GraphX abstraction, which represents graphs as horizontally-partitioned collections and graph computation as dataflow operators on those collections. Not only does GraphX support existing graph-parallel abstractions and a wide range of iterative graph algorithms, it enables the composition of graphs and collections, freeing the user to adopt the most natural view without concern for data movement or duplication.

Guided by the connection between graph computation and dataflow operators, we recast recent advances in graph processing systems as range of classic optimizations in database systems. We recast vertex-cut graph partitioning as horizontally-partitioned vertex and edge collections, active vertex tracking as incremental view maintenance, and vertex mirroring as multicast joins with routing tables. As a result, for graph algorithms, GraphX is over an order of magnitude faster than the base dataflow system and is comparable to or faster than specialized graph processing systems. In addition, GraphX benefits from features provided by recent dataflow systems such as low-cost fault tolerance and transparent recovery.

We believe that our work on GraphX points to a larger research agenda in the unification of specialized data processing systems. Recent advances in specialized systems for topic modeling, graph processing, stream processing, and deep learning have revealed a range of new system optimizations and design trade-offs. However, the full potential of these systems is often realized in their integration (e.g., applying deep learning to text and images in a social network). By casting these systems within a common paradigm (e.g., dataflow operators) we may reveal common patterns and enable new analytics capabilities.

9 Acknowledgments

We would like to thank Matei Zaharia, Peter Bailis, and our colleagues in the AMPLab, Databricks, and GraphLab for their help in building and presenting the GraphX system. We also thank the OSDI reviewers and our shepherd Frans Kaashoek for their insightful comments and guidance in preparing this paper. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adobe, Apple, Inc., Bosch, C3Energy, Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, and Yahoo!.

[2] `https://spark.apache.org`

[3] For a large-scale commercial use case see [14].

References

[1] ABADI, D. J., MARCUS, A., MADDEN, S. R., AND HOLLENBACH, K. SW-Store: A vertically partitioned DBMS for semantic web data management. *PVLDB 18*, 2 (2009), 385–406.

[2] AFRATI, F. N., AND ULLMAN, J. D. Optimizing joins in a map-reduce environment. In *EDBT* (2010), pp. 99–110.

[3] BLANAS, S., PATEL, J. M., ERCEGOVAC, V., RAO, J., SHEKITA, E. J., AND TIAN, Y. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 975–986.

[4] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW* (2011), pp. 587–596.

[5] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *WWW'04*.

[6] BROEKSTRA, J., KAMPMAN, A., AND HARMELEN, F. V. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proceedings of the First International Semantic Web Conference on The Semantic Web* (2002), ISWC '02, pp. 54–68.

[7] BULUÇ, A., AND GILBERT, J. R. The combinatorial BLAS: design, implementation, and applications. *IJHPCA 25*, 4 (2011), 496–509.

[8] ÇATALYÜREK, U. V., AYKANAT, C., AND UÇAR, B. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput. 32*, 2 (2010), 656–683.

[9] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys* (2012), pp. 85–98.

[10] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *OSDI* (2004).

[11] EWEN, S., TZOUMAS, K., KAUFMANN, M., AND MARKL, V. Spinning fast iterative data flows. *Proc. VLDB 5*, 11 (July 2012), 1268–1279.

[12] FEIGE, U., HAJIAGHAYI, M., AND LEE, J. R. Improved approximation algorithms for minimum-weight vertex separators. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2005), STOC '05, ACM, pp. 563–572.

[13] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI'12, USENIX Association, pp. 17–30.

[14] HUANG, A., AND WU, W. Mining ecommerce graph data with spark at alibaba taobao. http://databricks.com/blog/2014/08/14/mining-graph-data-with-spark-at-alibaba-taobao.html, 2014.

[15] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007), pp. 59–72.

[16] KARYPIS, G., AND KUMAR, V. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput. 48*, 1 (1998), 96–129.

[17] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale graph computation on just a PC. In *OSDI* (2012).

[18] LESKOVEC, J., LANG, K. J., DASGUPTA, A., , AND MAHONEY, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6*, 1 (2008), 29–123.

[19] LOW, Y., ET AL. GraphLab: A new parallel framework for machine learning. In *UAI* (2010), pp. 340–349.

[20] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB* (2012).

[21] MACKERT, L. F., AND LOHMAN, G. M. R* optimizer validation and performance evaluation for distributed queries. In *VLDB'86* (1986), pp. 149–159.

[22] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *SIGMOD* (2010), pp. 135–146.

[23] MANOLA, F., AND MILLER, E. RDF primer. *W3C Recommendation 10* (2004), 1–107.

[24] MONDAL, J., AND DESHPANDE, A. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 145–156.

[25] MURRAY, D. Building new frameworks on Naiad. blog post: http://bigdataatsvc.wordpress.com/2014/04/29/ building-new-frameworks-for-naiad/, April 2014.

[26] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *SOSP '13*.

[27] NAJORK, M., FETTERLY, D., HALVERSON, A., KENTHAPADI, K., AND GOLLAPUDI, S. Of hammers and nails: An empirical comparison of three paradigms for processing large graphs. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining* (2012), WSDM '12, ACM, pp. 103–112.

[28] NEUMANN, T., AND WEIKUM, G. RDF-3X: A RISC-style engine for RDF. *VLDB'08*.

[29] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. *SIGMOD* (2008).

[30] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[31] PRUD'HOMMEAUX, E., AND SEABORNE, A. SPARQL query language for RDF. Latest version available as `http://www.w3.org/TR/rdf-sparql-query/`, January 2008.

[32] PUJOL, J. M., ERRAMILLI, V., SIGANOS, G., YANG, X., LAOUTARIS, N., CHHABRA, P., AND RODRIGUEZ, P. The little engine(s) that could: scaling online social networks. In *SIGCOMM* (2010), pp. 375–386.

[33] ROBINSON, I., WEBBER, J., AND EIFREM, E. *Graph Databases*. O'Reilly Media, Incorporated, 2013.

[34] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. SOSP '13, ACM, pp. 472–488.

[35] SAAD, Y. *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[36] STANTON, I., AND KLIOT, G. Streaming graph partitioning for large distributed graphs. Tech. Rep. MSR-TR-2011-121, Microsoft Research, November 2011.

[37] STUTZ, P., BERNSTEIN, A., AND COHEN, W. Signal/collect: graph algorithms for the (semantic) web. In *ISWC* (2010).

[38] UGANDER, J., AND BACKSTROM, L. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining* (New York, NY, USA, 2013), WSDM '13, ACM, pp. 507–516.

[39] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI'12.

Nail: A practical tool for parsing and generating data formats

Julian Bangert and Nickolai Zeldovich
MIT CSAIL

Abstract

Nail is a tool that greatly reduces the programmer effort for safely parsing and generating data formats defined by a grammar. Nail introduces several key ideas to achieve its goal. First, Nail uses a protocol grammar to define not just the data format, but also the internal object model of the data. Second, Nail eliminates the notion of semantic actions, used by existing parser generators, which reduces the expressive power but allows Nail to both *parse* data formats and *generate* them from the internal object model, by establishing a *semantic bijection* between the data format and the object model. Third, Nail introduces *dependent fields* and *stream transforms* to capture protocol features such as size and offset fields, checksums, and compressed data, which are impractical to express in existing protocol languages. Using Nail, we implement an authoritative DNS server in C in under 300 lines of code and grammar, and an `unzip` program in C in 220 lines of code and grammar, demonstrating that Nail makes it easy to parse complex real-world data formats. Performance experiments show that a Nail-based DNS server can outperform the widely used BIND DNS server on an authoritative workload, demonstrating that systems built with Nail can achieve good performance.

1 Introduction

Code that handles untrusted inputs, such as processing network data or parsing a file, is error-prone and is often exploited by attackers. For example, the libpng image decompression library had 24 remotely exploitable vulnerabilities from 2007 to 2013 [5], Adobe's PDF and Flash viewers have been notoriously plagued by input processing vulnerabilities, and even the zlib compression library had input processing vulnerabilities in the past [6]. With a memory-unsafe language like C, mistakes in input processing code can lead to memory errors like buffer overflows, and even with a memory-safe language like Java, inconsistencies between different parsers can lead to security issues [13].

A promising approach to avoid these problems is to specify a precise grammar for the input data format, and to use a parser generator, such as `lex` and `yacc`, to synthesize the input processing code. Developers that use a parser generator do not need to write error-prone input processing code on their own, and as long as the parser generator is bug-free, the application will be safe from input processing vulnerabilities. Grammars can also be re-used between applications, further reducing effort and eliminating inconsistencies.

Unfortunately, applying this approach in practice, using state-of-the-art parser generators, still requires too much manual programmer effort, and is still error-prone, for four reasons:

First, parser generators typically parse inputs into an abstract syntax tree (AST) that corresponds to the grammar. In order to produce a data structure that the rest of the application code can easily process, application developers must write explicit *semantic actions* that update the application's internal representation of the data based on each AST node. Writing these semantic actions requires the programmer to describe the structure of the input three times—once to describe the grammar, once to describe the internal data structure, and once again in the semantic actions that translate the grammar into the data structure—leading to potential bugs and inconsistencies. Furthermore, in a memory-unsafe language like C, these semantic actions often involve error-prone manual memory allocation and pointer manipulation.

Second, applications often need to produce output in the same format as their input—for example, applications might both read and write files, or both receive and send network packets. Most parser generators focus on just parsing an input, rather than producing an output, thus requiring the programmer to manually construct outputs, which is work-intensive and leads to more code that could contain errors. Some parser generators, such as Boost.Spirit [8], allow reusing the grammar for generating output from the internal representation. However, those generators require yet another set of semantic actions to be written, transforming the internal representation back into an AST.

Third, many data formats contain redundancies, such as repeating information in multiple structures. Applications usually do not explicitly check for consistency, and if different applications use different instances of the same value, an attacker can craft an input that causes inconsistencies [38]. Furthermore, security vulnerabilities can occur when an application assumes two repetitions of the same data to be consistent, such as allocating a buffer based on the value of one size field and copying into that buffer based on the value of another [28].

Finally, real-world data formats, such as PNG or PDF,

are hard to represent with existing parser generators. Those parsers cannot directly deal with length or checksum fields, so the programmer has to either write potentially unsafe code to deal with such features, or build contrived grammar constructs, such as introducing one grammar rule for each possible value of a length field. Offset fields, which specify the position at which some data structure is located, usually require the programmer to manipulate a parser's internal state to re-position its input. More complicated transformations, such as handling compressed data, cannot be represented at all.

This paper presents the design and implementation of Nail, a parser generator that greatly reduces the programmer effort required to use grammars. Nail addresses the above challenges with several key ideas, as follows.

First, Nail grammars define both a format's external representation and an internal object model. This removes the semantic actions and type declarations that programmers have to write with existing parser generators. While this somewhat reduces the flexibility of the internal model, it forces the programmer to clearly separate syntactic validation and semantic processing.

Second, this well-defined internal representation allows Nail to establish a *semantic bijection* between data formats and their internal object model. As a result, this enables Nail to not just parse input but also generate output from the internal representation, without requiring the programmer to write additional code.

Third, Nail introduces two abstractions, *dependent fields* and *transformations*, to elegantly handle problematic structures, such as offset fields or checksums. Dependent fields capture fields in a protocol whose value depends in some way on the value or layout of other parts of the format; for example, offset or length fields, which specify the position or length of another data structure, fall in this category. Transformations allow the programmer to escape the generated code to modify the raw data and interact with dependent fields in a controlled manner.

To evaluate whether Nail's design is effective at handling real-world data formats, we implemented a prototype of Nail for C. Using our prototype, we implemented grammars for parts of an IP network stack, for DNS packets, and for ZIP files, each in about a hundred lines of grammar. On top of these grammars, we were able to build a DNS server in under 200 lines of C code, and an `unzip` utility in about 50 lines of C code, with performance comparable to or exceeding existing implementations. This suggests both that Nail is effective at handling complex real-world data formats, and that Nail makes it easy for application developers to parse and generate external data representations. Performance results show that the Nail-based DNS server outperforms the widely used BIND DNS server, demonstrating that Nail-based parsers and generators can achieve good performance.

The rest of this paper is organized as follows. §2 puts Nail in the context of related work. §3 motivates the need for Nail by examining past data format vulnerabilities. §4 describes Nail's design. §5 discusses our implementation of Nail. §6 provides evaluation results, and §7 concludes.

2 Related work

Parsers. Generating parsers and generators from an executable specification is the core concept of interface generators, such as CORBA [30], XDR [37], and Protocol Buffers [40]. However, interface generators do not allow the programmer to specify the byte-level data format; instead, they define their own data encoding that is specific to a particular interface generator. For instance, XDR-based protocols are incompatible with Protocol Buffers. Moreover, this means that interface generators cannot be used to interact with existing protocols that were not defined using that interface generator in the first place. As a result, interface generators cannot be used to parse or generate widely used formats such as DNS or ZIP, which is a goal for Nail.

Closely related work has been done in the field of data description languages, for example PacketTypes [24] and DataScript [1]; a broader overview of data description languages can be found in Fisher et al [10]. PacketTypes implements a C-like structure model enhanced with length fields and constraints, but works only as a parser, and not as an output generator. DataScript adds output generation and built-in support for offset fields. A particularly sophisticated data description language, PADS [9], which is targeted more towards offline analysis, even features built-in support for XML and automatic grammar inference. However, these systems cannot easily handle complicated encodings such as compressed data, which are supported by Nail's stream transforms. While sophisticated languages like PADS allow for handling particular variations of offset fields, compressed data, or even XML entities, each of these features has to be implemented in the data description language and all associated tools. Nail's transformations keep the core language small, while enabling the wide range of features real-world protocols require.

Recently, the Hammer project [32] introduced a security-focused parser framework for binary protocols. Hammer implements grammars as language-integrated parser combinators, an approach popularized by Parsec for Haskell [21]. The parser combinator style (to our knowledge, first described by Burge [4]) is a natural way of concisely expressing top-down grammars [7] by composing them from one or multiple sub-parsers.[1] Hammer then constructs a tree of function pointers which can be invoked to parse a given input into an AST.

[1]For more background on the history of expressing grammars, see Bryan Ford's masters thesis [11], which also describes the default parsing algorithm used by Hammer.

Nail improves upon Hammer in three ways. First, Nail generates output in addition to parsing input. Second, Nail does not require the programmer to write potentially insecure semantic actions. Last, Nail's structural dependencies and stream transforms allow it to work with protocols that Hammer cannot handle, such as protocols with offset fields, length fields, checksums, or compressed data, although Hammer has special facilities for arrays immediately preceded by their length.

Parsifal [22] is a parser framework for OCaml that also supports generating output. Parsifal structures grammars as an OCaml type that holds an internal model and functions for parsing input and output. However, Parsifal can produce parsers and generators only for simple, fixed-size structures. The programmer can then use these when implementing parsers and generators for more complicated formats, manually handling offsets, checksums, and the like, risking bugs. Nail handles more complicated constructs without the programmer manually writing code to support them.

We presented an earlier design of Nail at a workshop [2]. At that stage, Nail had only limited support for dependent fields, and did not support stream transforms at all, which are crucial for supporting real-world formats like DNS and ZIP. The workshop paper also did not provide a detailed design discussion or evaluation.

Application use of parsers. Generated parsers have long been used to parse human input, such as programming languages and configuration files. Frequently, such languages are specified with a formal grammar in an executable form. Unfortunately, parser frameworks are seldom used to recognize machine-created input; as we demonstrate in §6, state-of-the-art parser generators are not suitable for parsing or generating many real-world data formats.

A notable exception is the Mongrel web server [36] which uses a grammar for HTTP written in the Ragel regular expression language [39]. Mongrel was re-written from scratch multiple times to achieve better scalability and design, yet the grammar was reused across all iterations [31]. We hope that Nail's ideas make it possible to handle a wider range of protocols using parser generators, and to build more applications on top of grammar-based parsers.

3 Motivation

To motivate the need for Nail, this section presents a case study of vulnerabilities due to ad-hoc input parsing and output generation. Broadly speaking, parsing vulnerabilities can lead to two kinds of problems—memory corruption and logic errors—and as we show, both are prevalent in software and lead to significant security problems.

Widely exploited parsing errors. Three recent high-profile security vulnerabilities are due to logic errors in input processing. In all cases, when the vulnerabilities were fixed, a similar flaw was exposed immediately afterwards, showing the need for a different approach to input handling that eliminates those vulnerabilities by design.

The Evasi0n jailbreak for iOS 6 [38] relies on the XNU kernel and user-mode code-signing verifier interpreting executable metadata differently, so the code signature checker sees different bytes at a virtual address than what the kernel maps into the process. The next version of iOS added an explicit check for this particular metadata inconsistency. However, because parsing and processing of the input data is still mixed, the jailbreakers could set a flag that re-introduced the inconsistency after the check, but before signatures are verified [16], which allowed iOS 7 to be jailbroken.

Similarly, vulnerabilities in X.509 parsers for SSL certificates allowed attackers to get certificates for domains they do not control. First, Moxie Marlinspike discovered that the X.509 parsers in popular browsers handle NUL-bytes in certificates incorrectly [23]. After this vulnerability was fixed, Dan Kaminsky discovered [20] that other structures, such as length fields and duplicated data, were also handled incorrectly.

Similarly, the infamous Android master key bug [13] completely bypassed Android security by exploiting parser inconsistencies between the ZIP handler that checks signatures for privileged applications and the ZIP implementation that ultimately extracts those files. Thus, privileged application bundles could be modified to include malicious code without breaking their signatures. Google quickly fixed this particular parser inconsistency, but another vulnerability, based on a different inconsistency between the parsers, was quickly disclosed [14].

Case study: ZIP file handling. To understand the impact of parsing mistakes in real-world software, we conducted a systematic study of vulnerabilities related to ZIP file parsing. The ZIP format has been associated with many vulnerabilities, and the PROTOS Genome project [34] found numerous security vulnerabilities related to input handling in most implementations of ZIP and other archive formats. We extend this study by looking at the CVE database.

We found 83 vulnerabilities in the CVE database [25] that mention the search string "ZIP." Just 16 of these vulnerabilities were related to processing ZIP archives; the rest were unrelated to ZIP archives or involved applications insecurely using the contents of untrusted ZIP files. Figure 1 summarizes the 16 ZIP-related vulnerabilities.

These input-processing vulnerabilities fall into two broad classes. The first class, which occurred 11 times,[2]

[2] We classified the following vulnerabilities as memory corruption

Classification	Example CVE	Example description	Count
Memory corruption	CVE-2013-5660	Buffer overflow	11
Parsing inconsistency	CVE-2013-1462	Multiple virus scanners interpret ZIP files incorrectly	4
Semantic misunderstanding	CVE-2014-2319	Weak cryptography used even if user selects AES	1
Total of all vulnerabilities related to .zip processing			**16**

Figure 1: Classification of vulnerabilities in the CVE database from 2010 to May 2014 containing the term "ZIP" and involving the ZIP file format.

is memory safety bugs, such as buffer overflows, which allow an adversary to corrupt the application's memory using specially crafted inputs. These mistakes arise in lower-level languages that do not provide memory safety guarantees, such as C, and can be partially mitigated by a wide range of techniques, for example static analysis, dynamic instrumentation, and address space layout randomization, that make it more difficult for an adversary to exploit these bugs. Nail helps developers using lower-level languages to avoid these bugs in the first place.

The second class, which occurred four times in our study, is logic errors, where application code misinterprets input data. Safe languages and exploit mitigation technologies do not help against such vulnerabilities. This can lead to serious security consequences when two systems disagree on the meaning of a network packet or a signed message, as shown by the vulnerabilities we described before. CVE-2013-0211 shows that logic errors can be the underlying cause of memory corruption, when one part of a parser interprets a size field as a signed integer and another interprets it as an unsigned integer. CVE-2013-7338 is a logic error that allows an attacker to craft ZIP files that are incorrectly extracted or result in application hangs with applications using a Python ZIP library, because this library does not check that two fields that contain the size of a file contain the same value. The Android ZIP file signature verification bug that we described earlier was also among these 4 vulnerabilities.

These mistakes are highly application-specific, and are difficult to mitigate using existing techniques, and these mistakes can occur even in high-level languages that guarantee memory safety. By allowing developers to specify their data format just once, Nail avoids logic errors and inconsistencies in parsing and output generation.

4 Design

Nail's goals are to reduce programmer effort required to safely interact with data formats and prevent vulnerabilities like those described in §3. In particular, this means:

- Using a single grammar to define both the *external format* and the *internal representation*. This allows the same grammar to be re-used in multiple programs, and helps avoid vulnerabilities like the Android Master Key bug.
- Parsing inputs into internal representations, as well as generating outputs from internal representations, without requiring the programmer to write any semantic actions. This prevents vulnerabilities such as the iOS XNU bug, where format recognition and semantics are mixed and interact in unexpected ways.
- Eliminating redundancy in internal representations, such as storing both an explicit length field and an implicit length of a container data structure, to provide programmers a consistent, unambiguous view of the data. This helps avoid bugs such as the one discovered in the Python ZIP library [28].
- Allow programmers to define grammars for complex real-world data formats through well-defined extensibility mechanisms. This helps prevent programmers from falling back on manual parsing when encountering a complex data format.

4.1 Overview

Internal model. Nail grammars describe both the *external format* and an *internal representation* of a protocol. Nail produces the following from a single, descriptive grammar:

- *Type declarations* for the internal model, which the application should use to represent data items in memory.
- The *parser*, a function that the application should invoke to parse a sequence of bytes into an instance of the above model.
- The *generator*, a function that the application should invoke to create a sequence of bytes from an instance of the model.

For example, Figure 2 shows a Nail grammar for DNS packets. For this grammar, Nail produces the type declarations shown in Figure 3, and the parser and generator functions shown in Figure 4.

attacks based on their description: CVE-2013-5660, -0742, -0138, CVE-2012-4987, -1163, -1162, CVE-2011-2265, CVE-2010-4535, -1657, -1336, and -1218.

```
1  dnspacket = {
2    id uint16
3    qr uint1
4    opcode uint4
5    aa uint1
6    tc uint1
7    rd uint1
8    ra uint1
9    uint3 = 0
10   rcode uint4
11   @qc uint16
12   @ac uint16
13   @ns uint16
14   @ar uint16
15   questions n_of @qc question
16   responses n_of @ac answer
17   authority n_of @ns answer
18   additional n_of @ar answer
19 }
20 question = {
21   labels compressed_labels
22   qtype uint16 | 1..16
23   qclass uint16 | [1,255]
24 }
25 answer = {
26   labels compressed_labels
27   rtype uint16 | 1..16
28   class uint16 | [1]
29   ttl uint32
30   @rlength uint16
31   rdata n_of @rlength uint8
32 }
33 compressed_labels = {
34   $decompressed transform dnscompress ($current)
35   labels apply $decompressed labels
36 }
37 label = { @length uint8 | 1..64
38            label n_of @length uint8 }
39 labels = <many label; uint8 = 0>
```

Figure 2: Nail grammar for DNS packets, used by our prototype DNS server.

```
struct dnspacket {
  uint16_t id;
  uint8_t qr;
  /* ... */
  struct {
    struct question *elem;
    size_t count;
  } questions;
};
```

Figure 3: Portions of the C data structures defined by Nail for the DNS grammar shown in Figure 2.

```
struct dnspacket *parse_dnspacket(NailArena *arena,
  const uint8_t *data,
  size_t size);

int gen_dnspacket(NailArena *tmp_arena,
  NailStream *out,
  struct dnspacket *val);
```

Figure 4: The API functions generated by Nail for parsing inputs and generating outputs for the DNS grammar shown in Figure 2.

Semantic bijection. Parsing inputs and generating outputs suggests a bijection between external data and its internal representation. However, a bijection in the traditional sense often does not make sense for data formats. Consider a grammar for a text language that tolerates white space, or a binary protocol that tolerates arbitrarily long padding. Program semantics should be independent of the number of padding elements in the input, and Nail therefore does not expose that information to the programmer. We call such discarded fields *constants*. Similarly, programs should not necessarily preserve the layout of objects referred to by their offsets.

Therefore, Nail establishes only a *semantic bijection* between the external format and the internal model. That is, when Nail parses an input into an internal representation, and then generates output from that representation, the two byte streams (input and output) will have the same meaning (i.e., be interpreted equivalently by Nail). However, the byte streams might not be identical. If the grammar consists of a simple protocol without offset fields, constants, and the like, there is a conventional bijection between internal models and valid parser inputs.

Hiding redundant information. Nail's internal model is designed to hide unneeded and redundant information from the application. Nail introduces *dependent fields*, which contain data that can be computed during generation and need to be kept as additional state during parsing. Dependent fields are, for example, used to represent lengths, offsets, and checksums. If dependent fields were exposed in the internal model, information would be duplicated and inconsistent internal data structures could be produced when data is modified. For example, when using Nail to handle UDP packets, without dependent fields, programmers might forget to update checksum fields when they modify the payload data.

Parser extensions. Real-world protocols contain complicated ways of encoding data. Fully representing these in an intentionally limited model such as our parser language is impractical. Therefore, Nail introduces *transformations*, which allow arbitrary code by the programmer to interact with the parser and generator. Nail parsers and generators interact with data through an abstract stream, which allows reading and writing of bits and re-positioning. Transformations allow the programmer to write functions in a general-purpose language that consume streams and define new temporary streams, while also reading or writing the values of dependent fields.

Initial versions of Nail's design included a special combinator for handling offset fields, which consumed a dependent field and applied a parser at the offset specified therein. However, it proved impossible to foresee all the ways in which a protocol could encode an offset; for example, some protocols such as PDF and ZIP locate structures

Nail grammar	External format	Internal data type in C
uint4	4-bit unsigned integer	uint8_t
int32 | [1,5..255,512]	Signed 32-bit integer $x \in \{1,5..255,512\}$	int32_t
uint8 = 0	8-bit constant with value 0	/* empty */
optional int8 | 16..	8-bit integer ≥ 16 or nothing	int8_t *

many int8 | ![0]	A NULL-terminated string	struct {
		  size_t N_count;
		  int_t *elem;
		};

{
  hours uint8
  minutes uint8
}
	Structure with two fields	struct {
		  uint8_t hours;
		  uint8_t minutes;
		};

<int8='"'; *p*; int8='"'>	A value described by parser *p*, in quotes	The data type of *p*

choose {
  A = uint8 | 1..8
  B = uint16 | 256..
}
	Either an 8-bit integer between 1 and 8, or a 16-bit integer larger than 256	struct {
		  enum {A, B} N_type;
		  union {
		    uint8_t a;
		    uint16_t b;
		  };
		};

@valuelen uint16
value n_of @valuelen uint8
	A 16-bit length field, followed by that many bytes	struct {
		  size_t N_count;
		  uint8_t *elem;
		};

$data transform deflate($current @method)	Applies programmer-specified function to create new stream (§4.4)	/* empty */
apply $stream *p*	Apply parser *p* to stream $stream (§4.4)	The data type of *p*
foo = *p*	Define rule foo as parser *p*	typedef /* type of *p* */ foo;
* p	Apply parser *p*	Pointer to the data type of *p*

Figure 5: Syntax of Nail parser declarations and the formats and data types they describe.

by scanning for a magic number starting at the end of the file or at a fixed offset. In nested grammars, offsets are also not necessarily computed from the beginning of a file or packet. Nail's transformations allow the programmer to write arbitrary functions that can handle such structures and streams, which are a generic abstraction for input and output data that allow the decoded data to be integrated with the rest of the generated Nail parser.

4.2 Basics

A Nail parser defines both the structure of some external format and a data type to represent that format. Parsers are constructed by combinators over simpler parsers, an approach popularized by the Parsec framework [21]. We provide the most common combinators familiar from other parser combinator libraries, such as Parsec and Hammer [32] and extend them so they also describe a data type.

We present both a systematic overview of Nail's syntax with short examples in Figure 5, and explain our design in more detail below, using a grammar for the well-known DNS protocol as a running example (shown in Figure 2).

Rules. A Nail grammar consists of rules that assign a parser to a name. Rules are written as assignments, such as `ints = /*parser definition*/`, which defines a rule called `ints`. As we will describe later in §4.3 and §4.4, rules can optionally consume parameters. Rules can be invoked in a Nail grammar anywhere a parser can appear. Rule invocations act as though the body of the rule had been substituted in the code. If parameters appear, they are passed by reference.

Integers and constraints. Nail's fundamental parsers represent signed or unsigned integers with arbitrary lengths up to 64 bits. Note that is possible to define

parsers for sub-byte lengths, for example, the flag bits in the DNS message header, in lines 5 through 8.

The grammar can also constrain the values of an integer. Nail expresses constraints as a set of permissible values or value ranges. Extending the Nail language and implementation to support richer constraints languages would be relatively trivial, however we have found that the current syntax covers permissible values within existing protocols correctly and concisely.

Repetition. The `many` combinator takes a parser and applies it repeatedly until it fails, returning an array of the inner parser's results. In line 39 of the DNS grammar, a sequence of labels is parsed by parsing as many labels as possible, that is, until an invalid length field is encountered. The `sepBy` combinator additionally takes a constant parser, which it applies in between parsing two values, but not before parsing the first value or after parsing the last. This is useful for parsing an array of items delimited by a separator.

Structures. Nail provides a structure combinator with semantic labels instead of the sequence combinator that other parser combinator libraries use to capture structures in data formats. The structure combinator consists of a sequence of fields, typically consisting of a label and a parser that describes the contents of that field, surrounded by curly braces. Other field types will be described below. The syntax of the structure combinator is inspired by the Go language [15], with field names preceding their definition.

Constants. In some cases, not all bytes in a structure actually contain information, such as magic numbers or reserved fields. Those fields can be represented in Nail grammars by constant fields in structures. Constant fields do not correspond to a field in the internal model, but they are validated during parsing and generated during output. Constants can either have integer values, such as in line 9 of the DNS grammar, or string values for text-based protocols, e.g. `many uint8 = "Foo"`.

In some protocols, there might be many ways to represent the same constant field and there is no semantic difference between the different syntactic representations. Nail therefore allows repeated constants, such as `many (uint8=' ')`, which parses any number of space characters, or `|| uint8 = 0x90 || uint16 = 0x1F0F`, which parses two of the many representations for x86 NOP instructions, which are used as padding between basic blocks in an executable.

As discussed above, choosing to use these combinators on constant parsers weakens the bijection between the format and the data type, as there are multiple byte-strings that correspond to the same internal representation and the generator chooses one of these.

Wrap combinator. When implementing real protocols with Nail, we often found structures that consist of many constants and only one named field. This pattern is common in binary protocols which use fixed headers to denote the type of data structure to be parsed. In order to keep the internal representation cleaner, we introduced the wrap combinator, which takes a sequence of parsers containing exactly one non-constant parser. The external format is defined as though the wrap combinator were a structure, but the data model does not introduce a structure with just one element, making the application-visible representation (and thus application code) more concise. Line 39 of the DNS grammar uses the wrap combinator to hide the terminating NUL-byte of a sequence of labels.

Choices. If multiple structures can appear at a given position in a format, the programmer lists the options along with a label for each in the `choose` combinator. During parsing, Nail remembers the current input position and attempts each option in the order they appear in the grammar. If an option fails, the parser backtracks to the initial position. If no options succeed, the entire combinator fails. In the data model, choices are represented as tagged unions. The programmer has to be careful when options overlap, because if the programmer meant to generate output for a choice, but the external representation is also valid for an earlier, higher-priority option, the parser will interpret it as such. However, real data formats normally do not have this overlap and we did not encounter it in the grammars we wrote. An example is provided in Figure 6.

Optional. Nail includes an `optional` combinator, which attempts to recognize a value, but succeeds without consuming input when it cannot recognize that value. Syntactically, `optional` is equivalent to a choice between the parser and an empty structure, but in the internal model it is more concisely represented as a reference that is null when the parser fails. For example, the grammar for Ethernet headers uses `optional vlan_header` to parse the VLAN header that appears only in Ethernet packets transmitted to a non-default VLAN.

References. Rules allow for recursive grammars. To support recursive data types, we introduce the reference combinator `*` that does not change the syntax of the external format described, but introduces a layer of indirection, such as a reference or pointer, to the model data type. The reference combinator does not need to be used when another combinator, such as `optional` or `many`, already introduces indirection in the data type. An example is shown in Figure 6.

4.3 Dependent fields

Data formats often contain values that are determined by other values or the layout of information, such as checksums, duplicated information, or offset and length

```
expr = choose {
  PAREN = <uint8='('; *expr; uint8=')'>
  PRODUCT = sepBy1 uint8='*' expr
  SUM = sepBy1 uint8='+' expr
  INTEGER = many1 uint8 | '0' .. '9'
}
```

Figure 6: Grammar for sums and products of integers.

fields. We represent such values using *dependent fields* and handle them transparently during parsing and generation without exposing them to the internal model.

Dependent fields are defined within a structure like normal fields, but their name starts with an `@` symbol. A dependent field is in scope and can be referred to by the definition of all subsequent fields in the same structure. Dependent fields can be passed to rule invocations as parameters.

Dependent fields are handled like other fields when parsing input, but their values are not stored in the internal data type. Instead the value can be referenced by subsequent parsers and it discarded when the field goes out of scope. When generating output, Nail visits a dependent field twice. First, while generating the other fields of a structure, the generator reserves space for the dependent field in the output. Once the dependent field goes out of scope, the generator writes the dependent field's value to this space.

Nail provides only one built-in combinator that uses dependent fields, `n_of`, which acts like the `many` combinator, except it represents an exact number, specified in the dependent field, of repetitions, as opposed to as many repetitions as possible. For example, DNS labels, which are encoded as a length followed by a value, are described in line 38 of the DNS grammar. Other dependencies, such as offset fields or checksums, are not handled directly by combinators, but through transformations, as we describe next.

4.4 Input streams and transformations

Traditional parsers handle input one symbol at a time, from beginning to end. However, real-world formats often require non-linear parsing. Offset fields require a parser to move to a different position in the input, possibly backwards. Size fields require the parser to stop processing before the end of input has been reached, and perhaps resume executing a parent parser. Other cases, such as compressed data, require more complicated processing on parts of the input before it can be handled.

Nail introduces two concepts to handle these challenges, *streams* and *transformations*. Streams represent a sequence of bytes that contain some external format. The parsers and generators that Nail generates always operate on an implicit stream named `$current` that they process front to back, reading input or appending output. Grammars can use the `apply` combinator to parse or generate external data on a different stream, inserting the result in the data model.

Streams are passed as arguments to a rule or defined within the grammar through *transformations*. The current stream is always passed as an implicit parameter.

Transformations are two arbitrary functions called during parsing and output generation. The parsing function takes any number of stream arguments and dependent field values, and produces any number of temporary streams. This function may reposition and read from the input streams and read the values of dependent fields, but not change their contents and values. The generating function has to be an inverse of the parsing function. It takes the same number of temporary streams that the parsing function produces, and writes the same number of streams and dependent field values that the parsing function consumes.

Typically, the top level of most grammars is a rule that takes only a single stream, which may then be broken up by various transformations and passed to sub-rules, which eventually parse various linear fragment streams. Upon parsing, these fragment streams are generated and then combined by the transforms.

To reduce both programmer effort and the risk of unsafe operations, Nails provides implementations of transformations for many common features, such as checksums, size, and offset fields. Furthermore, Nail provides library functions that can be used to safely operate on streams, such as splitting and concatenation. Nail implements streams as iterators, so they can share underlying buffers and can be efficiently duplicated and split.

Transformations need to be carefully written, because they can violate Nail's safety properties and introduce bugs. However, as we will show in §6.2, Nail transformations are much shorter than hand-written parsers, and many formats can be represented with just the transformations in Nail's standard library. For example, our Zip transformations are 78 lines of code, compared to 1600 lines of code for a hand-written parser. Additionally, Nail provides convenient and safe interfaces for allocating memory and accessing streams that address the most common occurrences of buffer overflow vulnerabilities.

Transformations can handle a wide variety of patterns in data formats, including the following:

Offsets. A built-in transformation for handling offset fields, which is invoked as follows: `$fragment transform offset_u32($current, @offset)`. This transformation corresponds to two functions for parsing and generation, as shown in Figure 7. It defines a new stream `$fragment` that can be used to parse data at the offset contained in `@offset`, by using `apply $fragment some_parser`.

```
int offset_u32_parse(NailArena *tmp,
  NailStream *out_str, NailStream *in_current,
  const uint32_t *off)
{
  /* out_str = suffix of in_current
                at offset *off */
}

int offset_u32_generate(NailArena *tmp,
  NailStream *in_fragment,
  NailStream *out_current, uint32_t *off)
{
  /* *off = position of out_current */
  /* append in_fragment to out_current */
}
```

Figure 7: Pseudocode for two functions that implement the offset transform.

Sizes. A similar transformation handles size fields. Just like the offset transform, it takes two parameters, a stream and a dependent field, but instead of returning the suffix of the current stream after an offset, it returns a slice of the given size from the current stream starting at its current position. When generating, it appends the fragment stream to the current stream and writes the size of the fragment to the dependent field.

Compressed data. Encoded, compressed, or encrypted data can be handled transparently by writing a custom transformation that transforms a coded stream into one that can be parsed by a Nail grammar and vice versa. This transformation must be carefully written to not have bugs.

Checksums. Checksums can be verified and computed in a transformation that takes a stream and a dependent field. In some cases, a checksum is calculated over a buffer that contains the checksum itself, with the checksum being set to some particular value. Because the functions implementing a transformation are passed a pointer to any dependent fields, the checksum function can set the checksum's initial value before calculating the checksum over the entire buffer, including the checksum.

A real-world example with many different transforms, used to support the ZIP file format, is described in §6.1.

5 Implementation

The current prototype of the Nail parser generator supports the C programming language. The implementation parses Nail grammars with Nail itself, using a 130-line Nail grammar feeding into a 2,000-line C++ program that emits the parser and generator code. Bootstrapping is performed with a subset of the grammar implemented using conventional grammars. An option for C++ STL data models is in development. In this section, we will discuss some particular features of our parser implementation.

A generated Nail parser makes two passes through the input: the first to validate and recognize the input, and the second to bind this data to the internal model. Currently the parser uses a straightforward top-down algorithm, which can perform poorly on grammars that backtrack heavily. However, preparations have been made to add Packrat parsing [12] that achieve linear time even in the worst case.

Defense-in-depth. Security exploits often rely on raw inputs being present in memory [3], for example to include shell-code or crafted stack frames for ROP [29] attacks in padding fields or the application executing a controlled sequence of heap allocations and de-allocations to place specific data at predictable addresses [18, 19]. Because the rest of the application or even Nail's generated code may contain memory corruption bugs, Nail carefully handles memory allocations as defense-in-depth to make exploiting such vulnerabilities harder.

When parsing input, Nail uses two separate memory arenas. These arenas allocate memory from the system allocator in large, fixed-size blocks. Allocations are handled linearly and all data in the arena is zeroed and freed at the same time. Nail uses one arena for data used only during parsing, including dependent fields and temporary streams; this arena is released before the parser returns. The other arena is used to allocate the internal data type returned and is freed by the application once it is done processing an input.

Furthermore, the internal representation does not include any references to the input stream, which can therefore be zeroed immediately after the parser succeeds, so an attacker has to write an exploit that works without referencing data from the raw input.

Finally, Nail performs sophisticated error handling only in a special debug configuration and will print error messages about the input only to `stderr`. Besides complicating the parser, advanced error handling invites programmers to attempt to fix malformed input, such as adding reasonable defaults for a missing field. Such error-fixing not only introduces parser inconsistencies, but also might allow an attacker to sneak inconsistent input past a parser.

6 Evaluation

In our evaluation of Nail, we answer four questions:

- Can Nail grammars support real-world data formats, and are Nail's techniques critical to handling these formats?
- How much programmer effort is required to build an application that uses Nail for data input and output?
- Does using Nail for handling input and output improve application security?

Protocol	LoC	Challenging features
DNS packets	48+64	Label compression, count fields
ZIP archives	92+78	Checksums, offsets, variable length trailer, compression
Ethernet	16+0	—
ARP	10+0	—
IP	25+0	Total length field, options
UDP	7+0	Checksum, length field
ICMP	5+0	Checksum

Figure 8: Protocols, sizes of their Nail grammars, and challenging aspects of the protocol that cannot be expressed in existing grammar languages. A + symbol counts lines of Nail grammar code (before the +) and lines of C code for protocol-specific transforms (after the +).

- Does Nail achieve acceptable performance?

6.1 Data formats

To answer the first question, we used Nail to implement grammars for seven protocols with a range of challenging features. Figure 8 summarizes these protocols, the lines of code for their Nail grammars, and the challenging features that make the protocols difficult to parse with state-of-the-art parser generators. We find that despite the challenging aspects of these protocols, Nail is able to capture the protocols, by relying on its novel features: dependent fields, streams, and transforms. In contrast, state-of-the-art parser generators would be unable to fully handle 5 out of the 7 data formats. In the rest of this subsection, we describe the DNS and Zip grammars in more detail, focusing on how Nail's features enable us to support these formats.

DNS. In Section 4, we introduced Nail's syntax with a grammar for DNS packets, shown in Figure 2. The grammar corresponds almost directly to the diagrams in RFC 1035, which defines DNS [26: §4]. Each DNS packet consists of a header, a set of question records, and a set of answer records. Domain names in both queries and answers are encoded as a sequence of labels, terminated by a zero byte. Labels are Pascal-style strings, consisting of a length field followed by that many bytes comprising the label.

One challenging aspect of DNS packets lies in the count fields (`qc`, `ac`, `ns`, and `ar`), which represent the number of questions or answers in another part of the packet. Nail's `n_of` combinator handles this situation easily, which would have been difficult to handle for other parsers.

Another challenging aspect of DNS is label compression [26: §4.1.4]. Label compression is used to reduce the size overhead of including each domain name multiple

```
int dnscompress_parse(NailArena *tmp,
  NailStream *out_decomp,
  NailStream *in_current);

int dnscompress_generate(NailArena *tmp,
  NailStream *in_decomp,
  NailStream *out_current);
```

Figure 9: Signatures of stream transform functions for handling DNS label compression.

times in a DNS reply (once in the question section, and at least once in the response section). If a domain name suffix is repeated, instead of repeating that suffix, the DNS packet may write a two-bit marker sequence followed by a 14-bit offset into the packet, indicating the position of where that suffix was previously encoded.

Handling label compression in existing tools, such as Bison or Hammer, would be awkward at best, because some ad-hoc trick would have to be used to re-position the parser's input stream. Keeping track of the position of all recognized labels would not be enough, as the offset field may refer to any byte within the packet, not just the beginning of labels. For this reason, the DNS server used as the example for Hammer does not support compression.

In contrast, Nail is able to handle label compression, by using a stream transform; the signatures of the two transform functions are shown in Figure 9. When parsing a packet, this transform decompresses the DNS label stream by following the offset pointers. When generating a packet, this transform receives the current suffix as an input, and scans the packet so far for previous occurrences, which implements compression.

ZIP files. An especially tricky data format is the ZIP compressed archive format [33]. ZIP files are normally parsed end-to-beginning. At the end of each ZIP file is an *end-of-directory header*. This header contains a variable-length comment, so it has to be located by scanning backwards from the end of the file until a magic number and a valid length field is found. Many ZIP implementations disagree on how to find this header in confusing situations, such as when the comment contains the magic number [42].

This end-of-directory header contains the offset and size of the *ZIP directory*, which is an array of *directory entry headers*, one for every file in the archive. Each entry stores file metadata, such as file name, compressed and uncompressed size, and a checksum, in addition to the offset of a *local file header*. The local file header duplicates most information from the directory entry header. The compressed file contents follow the header immediately.

Duplicating information made sense when ZIP files were stored on floppy disks with slow seek times and high

```
zip_file = {
  $file, $header transform
    zip_end_of_directory ($current)
  contents apply $header
    end_of_directory ($file)
}
end_of_directory ($file) = {
  // ...
  @directory_size uint32
  @directory_start uint32
  $dirstr1 transform
    offset_u32 ($filestream @directory_start)
  $directory_stream transform
    size_u32 ($dirstr1 @directory_size)
  @comment_length uint16
  comment n_of @comment_length uint8
  files apply $directory_stream n_of
    @total_directory_records dir_entry ($file)
}
dir_entry ($file) = {
  // ...
  @compression_method uint16
  mtime uint16
  mdate uint16
  @crc32 uint32
  @compressed_size uint32
  @uncompressed_size uint32
  @file_name_len uint16
  @extra_len uint16
  @comment_len uint16
  // ...
  @off uint32
  filename n_of @file_name_len uint8
  extra_field n_of @extra_len uint8
  comment n_of @comment_len uint8
  $content transform offset_u32 ($file @off)
  contents apply $content
    file (@crc32, @compression_method,
          @compressed_size, @uncompressed_size)
}
file (@crc32 uint32, @method uint16,
      @compressed_size uint32,
      @uncompressed_size uint32) = {
  uint32 = 0x04034b50
  version uint16
  flags file_flags
  @method_lcl uint16
  // ...
  $compressed transform
    size_u32 ($current @compressed_size)
  $uncompressed transform
    zip_compression ($compressed @method)
  transform crc_32 ($uncompressed @crc32)
  contents apply $uncompressed many uint8
  transform u16_depend (@method_lcl @method)
  // ...
}
```

Figure 10: Nail grammar for ZIP files. Various fields have been cut for brevity.

fault rates, and memory constraints made it impossible to keep the ZIP directory in memory or the archive was split across multiple disks. However, care must be taken that the metadata is consistent. For example, vulnerabilities could occur if the length in the central directory is used to allocate memory and the length in the local directory is used to extract without checking that they are equal first, as was the case in the Python ZIP library [28]. Figure 10 shows an abbreviated version of our ZIP file grammar. The ZIP grammar is a good example of how transformations capture complicated syntax in a real-world file format; existing parser languages cannot handle a file format of this complexity.

The `zip_file` grammar first splits the entire file stream into two streams based on the `zip_end_of_directory` transform on line 2. The corresponding C function `zip_end_of_directory_parse` finds the end-of-directory header as described above, by scanning the file backwards, and splits the file into two streams, one containing the end-of-directory header and one containing the file contents. The `end_of_directory` rule is then applied to the header stream in line 4. All offsets in the ZIP file refer to the beginning of the file, so the stream `$file` which contains the file contents without the header is passed as an argument to all parsers from hereon.

The directory header contains the offset and size of the ZIP directory (lines 9 and 10). The `offset` and `size` transformations extract a stream containing just the directory from the file contents. This stream is then parsed as an array of directory entries in line 17. Each directory entry in turn points to a local file header, which is similarly extracted and parsed with the `file` rule.

The `file` rule starting at line 41, describing a ZIP file entry, takes dependent field parameters containing file metadata information from the directory header. However, this same information is duplicated in the file entry, so the grammar uses the Nail-supplied `u16_depend` transform to check whether the two values are equal. Unlike most other transforms, `u16_depend` does not consume or produce strings; it only checks that two dependent fields are equal when parsing, and assigns the value of the second field to the first when generating. This ensures that the programmer does not have to worry about inconsistencies when handling the internal representation of a ZIP file.

Immediately following the file entry is the compressed data. Because most compression algorithms operate on unbounded streams of data, Nail decompresses data in two steps. First, it isolates the compressed data from the rest of the stream by using the `size` transform, which operates on the current stream, meaning it will consume data starting at the current position of the parser in the input. Second, Nail invokes a custom `zip_compression` transform that implements the appropriate compression

Application	LoC w/ Nail	LoC w/o Nail
DNS server	295	683 (Hammer parser)
unzip	220	1,600 (Info-Zip)

Figure 11: Comparison of code size for three applications written in Nail,and a comparable existing implementation without Nail.

and decompression functions based on the specified compression method. These functions are otherwise oblivious to the layout or metadata of the file.

6.2 Programmer effort

To evaluate how much programmer effort is required to build an application that uses Nail, we implemented two applications—a DNS server and an `unzip` program—based on the above grammars, and compared code size with comparable applications that process data manually, using `sloccount` [41]. We also compare the code size of our DNS server to a DNS server written using the Hammer parsing framework, although it does not fully support DNS (e.g., it lacks label compression, among other things). Figure 11 summarizes the results.

DNS. Our DNS server parses a zone file, listens to incoming DNS requests, parses them, and generates appropriate responses. The DNS server is implemented in 183 lines of C, together with 48 lines of Nail grammar and 64 lines of C code implementing stream transforms for DNS label compression. In comparison, Hammer [32] ships with a toy DNS server that responds to any valid DNS query with a CNAME record to the domain "spargelze.it". Their server consists of 683 lines of C, mostly custom validators, semantic actions, and data structure definitions, with 52 lines of code defining the grammar with Hammer's combinators. Their DNS server does not implement label compression, zone files, etc. From this, we conclude that Nail leads to much more compact code for dealing with DNS packet formats.

ZIP. We implemented a ZIP file extractor in 50 lines of C code, together with 92 lines of Nail grammar and 78 lines of C code implementing two stream transforms (one for the DEFLATE compression algorithm with the help of the zlib library, and one for finding the end-of-directory header).

Because more recent versions of ZIP have added more features, such as large file support and encryption, the closest existing tool in functionality is the historic version 5.4 of the Info-Zip unzip utility [35] that is shipped with most Linux distributions. The entire unzip distribution is about 46,000 lines of code, which is mostly optimized implementations of various compression algorithms and other configuration and portability code. However, unzip isolates the equivalent of our Nail tool in the file `extract.c`, which parses the ZIP metadata and calls various decompression routines in other files. This file measures over 1,600 lines of C, which suggests that Nail is highly effective at reducing manual input parsing code, even for the complex ZIP file format.

6.3 Security

We use a twofold approach to evaluate the security of applications implemented with Nail. First, we analyze a list of CVE's related to the ZIP file format and argue how our ZIP tools based on Nail are immune against those vulnerability classes. Second, we present the results of fuzz-testing our DNS server.

ZIP analysis. In §3, we presented 15 input handling vulnerabilities related to ZIP files.

11 of these vulnerabilities involved memory corruption during input handling. Because Nail's generated code checks offsets before reading and does not expose any untrusted pointers to the application, it is immune to memory corruption attacks by design.

Nail also protects against parsing inconsistency vulnerabilities like the four others we studied. Nail grammars explicitly encode duplicated information such as the redundant length fields in ZIP that caused a vulnerability in the Python ZIP library. The other three vulnerabilities exist because multiple implementations of the same protocol disagree on some inputs. Hand-written protocol parsers are not very reusable, as they build application-specific data structures and are tightly coupled to the rest of the code. Nail grammars, however, can be re-used between applications, avoiding protocol misunderstandings.

DNS fuzzing. To provide additional assurance that Nail parsers are free of memory corruption attacks, we ran the DNS fuzzer provided with the Metasploit framework [27] on our DNS server, which sent randomly corrupted DNS queries to our server for 4 hours, during which it did not crash or trigger the stack or heap corruption detector.

6.4 Performance

To evaluate whether Nail-based parsers are compatible with good performance, we compare the performance of our DNS server to that of ISC BIND 9 release 9.9.5 [17], a mature and widely used DNS server. We simulate a load resembling that of an authoritative name server. First, we generate domain names consisting of one or two labels randomly selected from an English dictionary, and one label that is one of three popular top-level domains (`com`, `net`, and `org`). Second, we randomly selected 90% of these domains and created a zone file that mapped these domain names to `127.0.0.1`. Finally, we used the `queryperf` tool provided with BIND to query each domain between zero and three times, using a DNS server running on the local machine. We used a single core of an Intel i7-3610QM system with 12GB of RAM. The

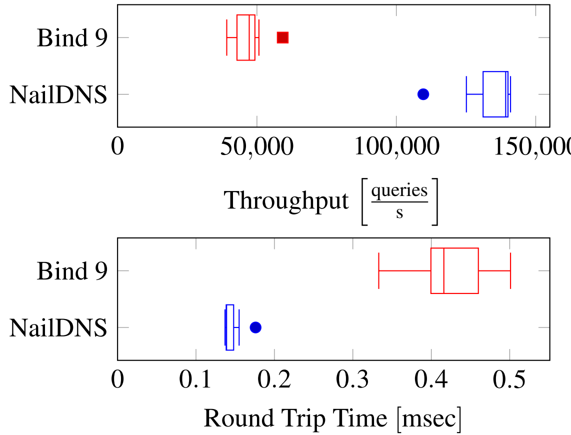


Figure 12: A box plot comparing the performance of the Nail-based DNS server compared to BIND 9.5.5 on 50,000 domains. The boxes show the interquartile range, with the middle showing the median result. The dots show outliers.

benchmark tool kept at most 20 queries outstanding at once, and was configured to repeat the same randomized sequence of queries for one minute. We repeated each test seven times with 50,000 domain names, restarting each daemon in between; we also repeated the tests with 1 million domain names, and found similar results. We also performed one initial dry run to warm the file system cache for the zone file.

The results are shown in Figure 12, and demonstrate that our Nail-based DNS server can achieve higher performance and lower latency than BIND. Although BIND is a more sophisticated DNS server, and implements many features that are not present in our Nail-based DNS server and that allow it to be used in more complicated configurations, we believe our results demonstrate that Nail's parsers are not a barrier to achieving good performance.

7 Conclusion

This paper presented the design and implementation of *Nail*, a tool for parsing and generating complex data formats based on a precise grammar. Nail helps programmers avoid memory corruption and inconsistency vulnerabilities while reducing effort in parsing and generating real-world protocols and file formats. Nail achieves this by reducing the expressive power of the grammar, establishing a *semantic bijection* between data formats and internal representations. Nail captures complex data formats by introducing *dependent fields*, *streams*, and *transforms*. Using these techniques, Nail is able to support DNS packet and ZIP file formats, and enables applications to handle these data formats in many fewer lines of code. Nail and all of the applications and grammars developed in this paper are released as open-source software, available at `https://github.com/jbangert/nail`.

Acknowledgments

We thank M. Frans Kaashoek and the anonymous reviewers for their feedback. This research was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] G. Back. Datascript - a specification and scripting language for binary data. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 66–77, Pittsburgh, PA, Oct. 2002.

[2] J. Bangert and N. Zeldovich. Nail: A practical interface generator for data formats. In *Proceedings of the 1st Workshop on Language-Theoretic Security (LangSec)*, pages 158–166, San Jose, CA, May 2014.

[3] S. Bratus, M. L. Patterson, and D. Hirsch. From "shotgun parsers" to more secure stacks. In *Shmoocon*, Nov. 2013.

[4] W. H. Burge. *Recursive programming techniques*. Addison-Wesley Reading, 1975.

[5] CVE Details. Libpng: Security vulnerabilities, 2014. `http://www.cvedetails.com/vulnerability-list/vendor_id-7294/Libpng.html`.

[6] CVE Details. GNU Zlib: List of security vulnerabilities, 2014. `http://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-1820/GNU-Zlib.html`.

[7] N. A. Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 285–296, Baltimore, MD, Sept. 2010.

[8] J. de Guzman and H. Kaiser. Boost Spirit 2.5.2, Oct. 2013. `http://www.boost.org/doc/libs/1_55_0/libs/spirit/doc/html/`.

[9] K. Fisher and R. Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 295–304, Chicago, IL, June 2005.

[10] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 2–15, Charleston, SC, Jan. 2006.

[11] B. Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.

[12] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, Oct. 2002.

[13] J. Freeman. Exploit (& fix) Android "master key", 2013. http://http://www.saurik.com/id/17.

[14] J. Freeman. Yet another Android master key bug, 2013. http://www.saurik.com/id/19.

[15] Google, Inc. *The Go Programming Language*, May 2014. http://golang.org/doc/.

[16] G. Hotz. evasi0n 7 writeup, 2013. http://geohot.com/e7writeup.html.

[17] Internet Systems Consortium. BIND 9 DNS server, 2014. http://www.isc.org/downloads/bind/.

[18] jp. Advanced Doug Lea's malloc exploits. *Phrack Magazine*, 11(61), Aug. 2003. http://phrack.org/issues/61/6.html.

[19] M. Kaempf. Vudo malloc tricks. *Phrack Magazine*, 11(57), Nov. 2001. http://phrack.org/issues/57/8.html.

[20] D. Kaminsky, M. L. Patterson, and L. Sassaman. PKI layer cake: New collision attacks against the global X.509 infrastructure. In *Proceedings of the 2010 Conference on Financial Cryptography and Data Security*, pages 289–303, Jan. 2010.

[21] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[22] O. Levillain. Parsifal: a pragmatic solution to the binary parsing problem. In *Proceedings of the 1st Workshop on Language-Theoretic Security (LangSec)*, pages 191–197, San Jose, CA, May 2014.

[23] M. Marlinspike. More tricks for defeating SSL in practice. https://www.blackhat.com/presentations/bh-usa-09/MARLINSPIKE/BHUSA09-Marlinspike-DefeatSSL-SLIDES.pdf, 2009. Black Hat USA.

[24] P. J. McCann and S. Chandra. Packet types: abstract specification of network protocol messages. *ACM SIGCOMM Computer Communication Review*, 30 (4):321–333, 2000.

[25] MITRE Corporation. Common vulnerabilities and exposures (CVE), 2014. http://http://cve.mitre.org/.

[26] P. Mockapetris. Domain names – implementation and specification. RFC 1035, Network Working Group, Nov. 1987.

[27] H. Moore et al. The metasploit project, 2014. http://www.metasploit.com/.

[28] Nandiya. zipfile - ZipExtFile.read goes into 100% CPU infinite loop on maliciously binary edited zips, Dec. 2013. http://bugs.python.org/issue20078.

[29] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11(58), Dec. 2001. http://phrack.org/issues/58/4.html.

[30] Object Management Group, Inc. CORBA FAQ, 2012. http://www.omg.org/gettingstarted/corbafaq.htm.

[31] M. Patterson. Langsec 2011-2016, May 2013. http://prezi.com/rhlij_momvrx/langsec-2011-2016/.

[32] M. Patterson and D. Hirsch. Hammer parser generator, Mar. 2014. https://github.com/UpstandingHackers/hammer.

[33] PKWARE, Inc. *.ZIP File Format Specification*, 6.3.3 edition, Sept. 2012. http://www.pkware.com/documents/casestudies/APPNOTE.TXT.

[34] PROTOS Project Consortium. PROTOS genome test suite c10-archive. Technical report, University of Oulu, 2007. https://www.ee.oulu.fi/research/ouspg/PROTOS_Test-Suite_c10-archive.

[35] G. Roelofs. Infozip, 1989. http://www.info-zip.org/.

[36] Z. Shaw. Mongrel HTTP server, 2008. http://www.rubyforge.org/projects/mongrel/.

[37] R. Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, Aug. 1995.

[38] Team Evaders. Swiping through modern security features. In *Proceedings of the HITB Amsterdam*, Apr. 2013.

[39] A. D. Thurston. Parsing computer languages with an automaton compiled from a single regular expression. In *Proceedings of the 11th International Conference on Implementation and Application of Automata*, pages 285–286, Taipei, Taiwan, 2006.

[40] K. Varda. Protocol buffers: Google's data interchange format, June 2008. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.

[41] D. A. Wheeler. Sloccount, 2014. http://www.dwheeler.com/sloccount/.

[42] J. Wolf. Stupid zip file tricks! In *BerlinSides 0x7DD*, 2013.

lprof: A Non-intrusive Request Flow Profiler for Distributed Systems

Xu Zhao, Yongle Zhang*, David Lion, Muhammad FaizanUllah, Yu Luo, Ding Yuan, Michael Stumm*
University of Toronto

Abstract

Applications implementing cloud services, such as HDFS, Hadoop YARN, Cassandra, and HBase, are mostly built as distributed systems designed to scale. In order to analyze and debug the performance of these systems effectively and efficiently, it is essential to understand the performance behavior of service requests, both in aggregate and individually.

lprof is a profiling tool that automatically reconstructs the execution flow of each request in a distributed application. In contrast to existing approaches that require instrumentation, *lprof* infers the request-flow entirely from runtime logs and thus does not require any modifications to source code. *lprof* first statically analyzes an application's binary code to infer how logs can be parsed so that the dispersed and intertwined log entries can be stitched together and associated to specific individual requests.

We validate *lprof* using the four widely used distributed services mentioned above. Our evaluation shows *lprof*'s precision in request extraction is 88%, and *lprof* is helpful in diagnosing 65% of the sampled real-world performance anomalies.

1 Introduction

Tools that analyze the performance behaviors of distributed systems are particularly useful; for example, they can be used to make more efficient use of hardware resources or to enhance the user experience. Optimizing performance can notably reduce data center costs for large organizations, and it has been shown that user response times have significant business impact [2].

In this paper, we present the design and implementation of *lprof*, a novel non-intrusive profiling tool aimed at analyzing and debugging the performance of distributed systems. *lprof* is novel in that (i) it does not require instrumentation or modifications to source code, but instead extracts information from the logs output during the course of normal system operation, and (ii) it is capable of automatically identifying, from the logs, each request and profile its performance behavior. Specifically, *lprof* is capable of reconstructing how each service request is processed as it invokes methods, uses helper threads, and invokes remote services on other nodes. We demonstrate that *lprof* is easy and practical to use, and that it is capable of diagnosing performance issues that existing solutions are not able to diagnose without instrumentation.

lprof outputs a database table with one line per request. Each entry includes (i) the type of the request, (ii) the starting and ending timestamps of the request, (iii) a list of nodes the request traversed along with the starting and ending timestamps at each node, and (iv) a list of the major methods that were called while processing the request. This table can be used to analyze the system's performance behavior; for example, it can be SQL-queried to generate *gprof*-like output [16], to graphically display latency trends over time for each type of service request, to graphically display average/high/low latencies per node, or to mine the data for anomalies. Section 2 provides a detailed example of how *lprof* might be used in practice.

Three observations led us to our work on *lprof*. First, existing tools to analyze and debug the performance of distributed systems are limited. For example, IT-level tools, such as Nagios [30], Zabbix [46], and OpsView [33], capture OS and hardware counter statistics, but do not relate them to higher-level operations such as service requests. A number of existing profiling tools rely on instrumentation; examples include gprof [16] that profiles applications by sampling function invocation points; MagPie [3], Project 5 [1], and X-Trace [14] that instrument the application as well as the network stack to monitor network communication; and commercial solutions such as Dapper [36], Boundary [5], and NewRelic [31]. As these tools require modifications to the software stack, the added performance overhead can be problematic for systems deployed in production. Recently, a number of tools applied machine learning techniques to analyze logs [29, 42], primarily to identify performance anomalies. Although such techniques can be effective in detecting individual anomalies, they often require separate correct and issue-laden runs, they do not relate anomalies to higher-level operations, and they are unable to detect *slowdown creep*.[1]

Our second observation is that performance analysis and debugging are generally given low priority in most

*Contributed equally to this paper.

[1] Slowdown creep is an issue encountered in organizations practicing agile development and deployment: each software update might potentially introduce some marginal additional performance overhead (e.g., <1%) that would not be noticeable in performance testing. However, with many frequent software releases, these individual slowdowns can add up to become significant over time.

organizations. This makes having a suitable tool that is easy and efficient to use more critical, and we find that none of the existing tools fit the bill. Performance analysis and debugging are given low priority for a number of reasons. Most developers prefer generating new functionality or fixing functional bugs. This behavior is also encouraged by aggressive release deadlines and company incentive systems. Investigating potential performance issues is frequently deferred because they can often easily be hidden by simply adding more hardware due to the horizontal scalability of these systems. Moreover, understanding the performance behavior of these systems is hard because the service is (i) distributed across many nodes, (ii) composed of multiple sub-systems (e.g., front-end, application, caching, and database services), and (iii) implemented with many threads/processes running with a high degree of concurrency.

Our third observation is that distributed systems implementing internet services tend to output a lot of log statements rich with useful information during their normal execution, even at the default verbosity.[2] Developers add numerous log output statements to allow for failure diagnosis and reproduction, and these statements are rarely removed [45]. This is evidenced by the fact that 81% of all statically found threads in HDFS, Hadoop Yarn, Cassandra, and HBase contains log printing statements of default verbosity in non-exception-handling code, and by the fact that Facebook has accumulated petabytes of log data [13]. In this paper we show that the information in the logs is sufficiently rich to allow the recovering of the inherent structure of the dispersed and intermingled log output messages, thus enabling useful performance profilers like *lprof*.

Extracting the per-request performance information from logs is non-trivial. The challenges include: (i) the log output messages typically consist of unstructured free-form text, (ii) the logs are distributed across the nodes of the system with each node containing the locally produced output, (iii) the log output messages from multiple requests and threads are intertwined within each log file, and (iv) the size of the log files is large.

To interpret and stitch together the dispersed and intertwined log messages of each individual request, *lprof* first performs static analysis on the system's bytecode. It analyzes each log printing statement to understand how to parse each output message and identifies the variable values that are output by the message. By further analyzing the data-flow of these variable values, static analysis extracts identifiers whose values remain unchanged in each specific request. Such identifiers can help associate log messages to individual requests. Since in practice an identifier may not exist in log messages or may not be not unique to each request, static analysis further captures the temporal relationships between log printing statements. Finally, static analysis identifies control paths across different local and remote threads. The information obtained from static analysis is then used by *lprof*'s parallel log processing component, which is implemented as a MapReduce [12] job.

The design of *lprof* has the following attributes:

- *Non-intrusive:* It does not modify any part of the existing production software stack. This makes it suitable for profiling *production systems*.
- *In-situ and scalable analysis:* The Map function in *lprof*'s MapReduce log processing job first stitches together the printed log messages from the same request *on the same node where the logs are stored*, which requires only one linear scan of each log file. Only summary information from the log file and only from requests that traverse multiple nodes is sent over the network in the shuffling phase to the reduce function. This avoids sending the logs over the network to a centralized location to perform the analysis, which is unrealistic in real-world clusters [27].
- *Compact representation allowing historical analysis:* *lprof* stores the extracted information related to each request in a compact form so that it can be retained permanently. This allows historical analysis where current performance behavior can be compared to the behavior at a previous point of time (which is needed to detect slowdown creep).
- *Loss-tolerant:* *lprof*'s analysis is not sensitive to the loss of data. If the logs of a few nodes are not available, *lprof* simply discards their input. At worst, this leads to some inaccuracies for the requests involving those nodes, but won't affect the analysis of requests not involving those nodes.

This paper makes the following contributions. First, we show that the standard logs of many systems contain sufficient information to be able to extract the performance behavior of any service-level request. Section 2 gives a detailed example of the type of information that is possible to extract from the logs and how this information can be used to diagnose and debug performance issues. Secondly, we describe the design and implementation of *lprof*. Section 3 provides a high-level overview, while Sections 4 and 5 describe details of *lprof*'s static analysis and how the logs are processed. Finally, Section 6 evaluates the techniques presented in this paper. We validated *lprof* using four widely-used distributed systems: HDFS, Hadoop YARN, Cassandra, and HBase. We show that *lprof* performs and scales well, and that it is able to

[2] This is in contrast to single-component servers that tend to limit log output [44]. Distributed systems typically output many log messages, in part because these systems are difficult to functionally debug, and in part because distributed systems, being horizontally scalable, are less sensitive to latency caused by the attendant I/O.

Request type	start timestamp	end timestamp	nodes traversed			log sequence ID
			IP	start time.	end time.	
writeBlock	2014-04-21 05:32:45,103	2014-04-21 05:32:47,826	172.31.9.26	05:32:45,103	05:32:47,826	41
			172.31.9.28	05:32:45,847	05:32:47,567	
			172.31.9.12	05:32:46,680	05:32:47,130	

Figure 1: One row of the request table constructed by *lprof* containing information related to one request. The "node traversed" column family [7] contains the IP address, the starting and ending timestamp on each node this request traversed. In this case, the HDFS writeBlock request traverses three nodes. The "log sequence ID" column contains a hash value that can be used to index into another table containing the sequence of log printing statements executed by this request.

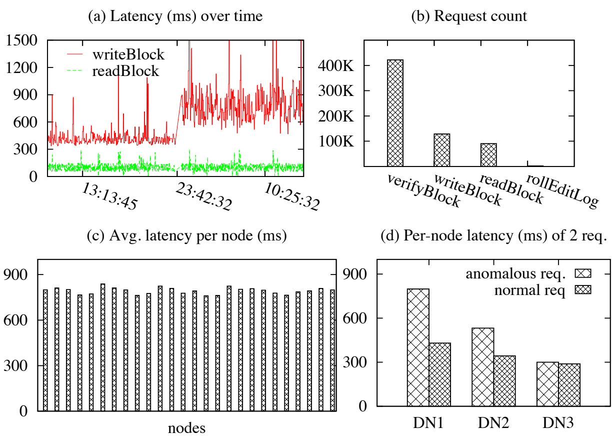


Figure 2: *lprof*'s analysis on HDFS' performance.

attribute 88% of all log messages to the correct requests. We discuss the limitations of *lprof* in Section 7 and close with related work and concluding remarks.

2 Motivating Example

To illustrate how *lprof*'s request flow analysis might be used in practice, we selected a performance issue reported by a (real) user [20] and reproduced the anomaly on a 25-node cluster.

In this example, an HDFS user suspects that the system has become slow after a software upgrade. Applying *lprof* to analyze the logs of the cluster produces a request table as shown in Figure 1. The user can perform various queries on this table. For example, she can examine trends in request latencies for various request types over time, or she can count the number of times each request type is processed during a time interval. Figures 2 (a) and (b) show how *lprof* visualizes these results.[3]

Figure 2 (a) clearly shows an anomaly with writeBlock requests at around 23:42. A sudden increase in writeBlock's latency is clearly visible while the latencies of the other requests remain unchanged. The user might suspect this latency increase is caused by a few nodes that are "stragglers" due to an unbalanced workload or a network problem. To determine whether this is the case, the user compares the latencies of each writeBlock request after 23:42 across the different nodes. This is shown in Figure 2 (c), which suggests no individual node is abnormal.

The user might then want to compare a few single requests before and after 23:42. This can be done by selecting corresponding rows from the database and comparing the per-node latency between an anomalous request and a healthy one. Figure 2 (d) visualizes the latency incurred on different nodes for two write requests: one before 23:42 (healthy) and the other after (anomalous). The figure shows that for both requests, latency is highest on the first node and lowest on the third node. HDFS has each block replicated on three data nodes (DNs), and each writeBlock request is processed as a pipeline across the three DNs: DN1 updates the local replica, sends it to DN2, and only returns to the user after DN2's response is received. Therefore the latency of DN2 includes the latency on DN3 plus the network communication time between DN2 and DN3.

The figure also shows that the latency of one request is clearly higher than the latency of the second request on the first two DNs. This leads to the hypothesis that code changes are responsible for the latency increase. The HDFS cluster was indeed upgraded between the servicing of the two requests (from version 2.0.0 to 2.0.2). The log sequence identifier is then used to identify the code path taken by both requests, and a diff on the two versions of the source code reveals that an extra socket write between DNs was introduced in version 2.0.2. The HDFS developers later fixed this performance issue by combining both socket writes into one [20].

Figure 2 (b) shows another performance anomaly: the number of verifyBlock requests is suspiciously high. Further queries on the request database suggest that before the upgrade, verifyBlock requests appear once every 5 seconds on *every* datanode, generating a lot of log messages, while after the upgrade, they appear only rarely. Interestingly, we noticed this accidentally in our experiments. Clearly *lprof* is useful in detecting and diagnosing this case as well.

[3] We envision that *lprof* is run periodically to process the log messages generated since its previous run, appending the new entries to the table and keeping them forever to enable historical analysis and debug problems like performance creep. If space is a concern, then instead of generating one table entry per request, *lprof* can generate one table entry per time interval and request type, each containing attendant statistical information (e.g., count, average/high/low timestamps, etc.).

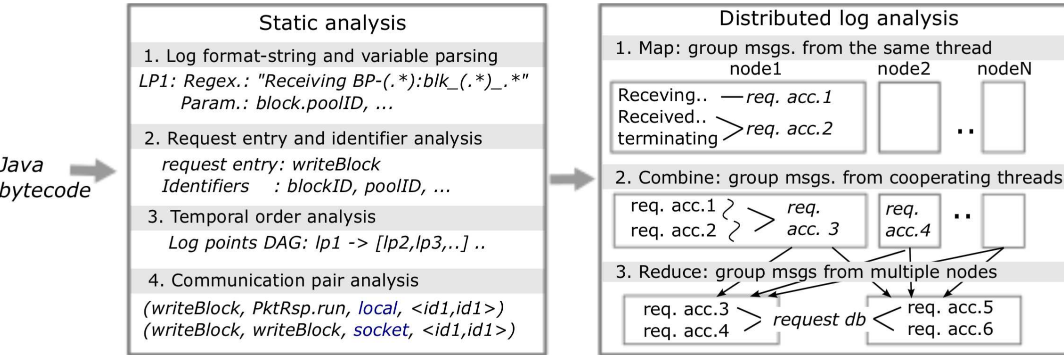


Figure 5: Overall architecture of *lprof*

```
1 class DataXceiver implements Runnable {
2   public void run() {
3     do { //handle one request per iteration
4       switch (readOpCode()) {
5       case WRITE_BLOCK: // a write request
7         writeBlock(proto.getBlock(), ..); break;
8       case READ_BLOCK: // a read request
9         readBlock(proto.getBlock(), ..); break;
10      } //proto.getBlock: deserialize the request
11    } while (!socket.isClosed());
12  }
13  void writeBlock(ExtendedBlock block..) {
14    LOG.info("Receiving block " + block);
15    sender.writeBlock(block,..); //send to next DN
16    responder = new PacketResponder(block,..);
17    responder.start(); // create a thread that
18  }                     // handles the acks
19 }
20 /* PacketResponder handles the ack responses */
21 class PacketResponder implements Runnable {
22   public void run() {
23     ack.readField(downstream); //read ack
24     LOG.info("Received block " + block);
25     replyAck(upstream); //send an ack to upstream
26     LOG.info(myString + " terminating");
27   }
28 }
```

Figure 3: Code snippet from HDFS that handles write request.

1 HDFS_READ, blockid: ***BP-9..9:blk_5..7_1032*** — *read*
2 Receiving block ***BP-9..9:blk_5..7***_1032
3 Received block ***BP-9..9:blk_5..7***_1032 — *write 1*
4 Receiving block ***BP-9..9:blk_4..8***_2313 — *write 2*
5 PacketResponder: ***BP-9..9:blk_5..7***_1032 terminating
6 opWriteBlock ***BP-9..9:blk_4..8***_2314 received exception

Figure 4: Part of an HDFS log. Request identifiers are shown in bold. Note that the timestamp of each message is not shown.

3 Overview of *lprof*

In this Section, before describing *lprof*'s design, we first discuss the challenges involved in stitching log messages together that were output when processing a single request. For example, consider how HDFS processes a write request as shown in Figure 3. On each datanode, a `DataXceiver` thread uses a `while` loop to process each incoming request. If the op-code is `WRITE_BLOCK`, then `writeBlock()` is invoked at line 7. At line 15, `writeBlock()` sends a replication request to the next downstream datanode. At line 16 - 17, a new thread associated with `PacketResponder` is created to receive the response from the downstream datanode so that it can send its response upstream. Hence, this code might output log messages as shown in Figure 4. These six log messages alone illustrate two challenges encountered:

1. The log messages produced when processing a single writeBlock request may come from multiple threads, and multiple requests may be processed concurrently. As a result, the log output messages from different requests will be intertwined.

2. The log messages do not contain an identifying substring that is unique to a request. For example, block ID "BP-9..9:blk_5..7" can be used to separate messages from different requests that do not operate on the same block, but cannot be used to separate the messages of the read and the first write request because they operate on the same block. Unfortunately, identifiers unique to a request rarely exist in real-world logs. In Section 7, we further discuss how *lprof* could be simplified if there were a unique request identifier in every log message.

To address these challenges *lprof* first uses static analysis to gather information from the code that will help map each log message to the processing of a specific request, and help establish an order on the log messages mapped to the request. In a second phase, *lprof* processes the logs using the information obtained from the static analysis phase; it does this as a MapReduce job.

We now briefly give a brief overview of *lprof*'s static analysis and log processing, depicted in Figure 5.

3.1 Static Analysis

lprof's static analysis gathers information in four steps.

(1) Parsing the log string format and variables obtains the signature of each log printing statement found in the code. An output string is composed of string constants

and variable values. It is represented by a regular expression (e.g., "Receiving block BP-(.*):blk_(.*)_.*"), which is used during the log analysis phase to map a log message to a set of *log points* in the code that could have output the log message. We use the term *log point* in this paper to refer to a log printing statement in the code. This step also identifies the variables whose values are contained in the log message.

(2) Request identifier and request entry analysis are used to analyze the dataflow of the variables to determine which ones are modified. Those that are not modified are recognized as *request identifiers*. Request identifiers are used to separate messages from different requests; that is, two log messages with different request identifiers are guaranteed to belong to different requests. However, the converse is not true: two messages with the same identifier value may still belong to different requests (e.g., both of the "read" and the "write 1" requests in Figure 4 have same the block ID).

Identifying request identifiers without domain expertise can be challenging. Consider "BP-9..9:blk_5..7_1032" in Figure 4 that might be considered as a potential identifier. This string contains the values of three variables as shown in Figure 6: `poolID`, `blockID`, and `generationStamp`. Only the substring containing `poolID` and `blockID` is suitable as a request identifier for *writeBlock*, because `generationStamp` can have different values while processing the same request (as exemplified by the "write 2" request in Figure 4).

To infer which log points belong to the processing of the same request, *top-level methods* are also identified by analyzing when identifiers are modified. We use the term top-level method to refer to the first method of any thread *dedicated* to the processing of a single type of request. For example, in Figure 3 `writeBlock()` and `PacketResponder.run()` are top-level methods, but `DataXceiver.run()` is not because it processes multiple types of requests. We say that method `M` is log point *p*'s top-level method if `M` is a top-level method and *p* is reachable from `M`.

If *lprof* can identify `readBlock()` and `writeBlock()` as being two top-level methods for different types of requests, it can separate messages printed by `readBlock()` from the ones printed by `writeBlock()` even if they have the same identifier value. We identify the top-level methods by processing each method in the call-graph in bottom-up order: if a method *M* modifies many variables that have been recognized as request identifiers in its callee *M'*, then *M'* is recognized as a top-level method. The intuition behind this design is that programmers naturally log request identifiers to help debugging, and the modification of a frequently logged but rarely modified variable is likely not part of the processing of a specific request.

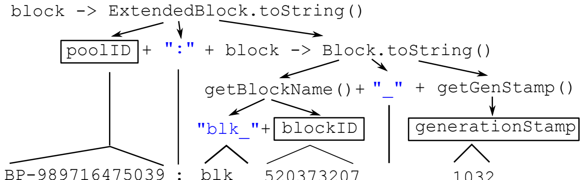


Figure 6: How *"BP-9..9:blk_5..7_1032"* is printed.

(3) Temporal order analysis is needed because there may not exist an ID unique to each request. For example, by inferring that line 26 is executed after line 24 in Figure 3, *lprof* can conclude that when two messages appear in the following order: "... terminating" and "Received block...", they cannot be from the same request even if they have the same block ID.

(4) Communication pair analysis is used to identify threads that communicate with each other. Log messages output by two threads that communicate could potentially be from processing of the same request. Such communication could occur through cooperative threads in the same process, or via sockets or RPCs across the network.

3.2 Distributed Log Analysis

The log analysis phase attributes each log message to a request, which is implemented using a MapReduce job. The map function groups together all log messages that were output by the same thread while processing the same request. A log message is added to a group if (i) it has the same top-level method, (ii) the request identifiers do not conflict, and (iii) the corresponding log point matches the temporal sequence in the control flow.

The reduce function merges groups if they represent log messages that were output by different threads when processing the same request. Two groups are merged if (i) the two associated threads could communicate, and (ii) the request identifiers do not conflict.

4 Static Analysis

lprof's static analysis works on Java bytecode. Each of the four steps in *lprof*'s static analysis is implemented as one analysis pass on the bytecode of the target system. We use the Chord static analysis framework [9]. For convenience, we explain *lprof* using examples in source code. All the information shown in the examples can be inferred from Java bytecode.

4.1 Parsing Log Printing Statements

This first step identifies every log point in the program. For each log point, *lprof* (i) generates a regular expres-

sion that matches the output log message, and (ii) identifies the variables whose values appear in the log output.

lprof identifies log points by searching for call instructions whose target method has the name `fatal`, `error`, `warn`, `info`, `debug`, or `trace`. This identifies all the logging calls if the system uses log4j [25] or SLF4J [37], two commonly used logging libraries that are used by the systems we evaluated.

To parse the format string of a log point into a regular expression, we use techniques similar to those used by two previous tools [42, 43]. We summarize the challenges we faced in implementing a log parser on real-world systems.

On the surface, parsing line 14 in Figure 3 into the regular expression "Receiving block (.*)", where the wildcard matches to the value of `block`, is straightforward. However, identifying the variables whose values are output at the log point is more challenging. In Java, the object's value is printed by calling its `toString()` method. Figure 6 shows how the value of `block` is eventually printed. In this case, *lprof* has to parse out the individual fields because only `poolID` and `blockID` are request identifiers, whereas `generationStamp` is modified during request processing. To do this, *lprof* recursively traces the object's `toString()` method and the methods that manipulate `StringBuilder` objects until it reaches an object of a primitive type.

For the HDFS log point above, the regular expression identified by *lprof* will be:

"Receiving block (.*):blk_(\d+)_(\d+)".

The three wildcard components will be mapped to `block.poolID`, `block.block.blockID`, and `block.block.generationStamp`, respectively.

lprof also needs to analyze the data-flow of any string object used at a log point. For example, `mystring` at line 26 in Figure 3 is a String object initialized earlier in the code. *lprof* analyzes its data-flow to identify the precise value of `mystring`.

Class inheritance and late binding in Java creates another challenge. For example, when a class and its super class both provide a `toString()` method, which one gets invoked is resolved only at runtime depending on the actual type of the object. To address this, *lprof* analyzes *both* classes' `toString()` methods, and generates *two* regular expressions for the one log point. During log analysis, if both regular expressions match a log message, *lprof* will use the one with the more precise match, i.e., the regular expression with a longer constant pattern.

4.2 Identifying Request Identifiers

This step identifies (i) request identifiers and (ii) top-level methods. We implement the inter-procedural analysis as

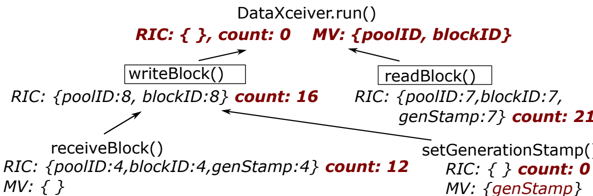


Figure 7: Request identifier analysis for the HDFS example of Figure 3. When analyzing `writeBlock()`, the request identifier candidate set (RIC) from its callee `receiveBlock()` is merged into its own set, so the cumulative count of `poolID` and `blockID` is increased to 8, 4 comes from `receiveBlock()` and 4 comes from the log points in `writeBlock()`. Since `generationStamp` is in `setGenerationStamp()`'s modified variable set (MV), it is removed from `writeBlock()`'s RIC set.

summary-based analysis [35]. It analyzes one method at a time and stores the result as the summary of that method. The methods are analyzed in bottom-up order along the call-graph and when a call instruction is encountered, the summary of the target method is used. Not being summary-based would require *lprof* to store the intermediate representation of the entire program in memory, which would cause it to run out of memory.

Data-flow analysis for request identifiers: *lprof* infers request identifiers by analyzing the inter-procedural data-flow of the logged variables. For each method *M*, *lprof* assembles two sets of variables as its summary: (i) *the request identifier candidate set* (RIC), which contains the variables whose values are output to a log and not modified by *M or its callees*, and (ii) *the modified variable set* (MV) which contains the variables whose values are modified. For each method *M*, *lprof* first initializes both sets to be empty. It then analyzes each instruction in *M*. When it encounters a log point, the variables whose values are printed (as identified by the previous step) are added to the RIC set. If an instruction modifies a variable *v*, *v* is added to the MV set and removed from the RIC set. If the instruction is a call instruction, *lprof* first merges the RIC and MV sets of the target method into the corresponding sets of the current method, and then, for each variable *v* in the MV set, *lprof* removes it from the RIC set if it contains *v*.

As an example, consider the following code snippet from `writeBlock()`:

```
1 LOG.info("Receiving " + block);
2 block.setGenerationStamp(latest);
```

The `setGenerationStamp()` method modifies the `generationStamp` field in `block`. In bottom-up order, *lprof* first analyzes `setGenerationStamp()` and adds `generationStamp` to its MV set. Later when *lprof* analyzes `writeBlock()`, it removes `generationStamp` from its RIC set because `generationStamp` is in the MV set of `setGenerationStamp()`.

Identifying top-level methods: the request identifier analysis stops at the root of the call-graph: either a thread entry method (i.e., `run()` in Java) or `main()`. However, a thread entry method might not be the entry of a service request. Consider the HDFS example shown in Figure 3. The `DataXceiver` thread uses a while loop to handle read and write requests. Therefore *lprof* needs to identify `writeBlock()` and `readBlock()` as the top-level methods instead of `run()`.

lprof identifies top-level methods by observing the propagation of variables in the RIC set and uses the following heuristic when traversing the call-graph bottom-up: if, when moving from a method *M* to its caller *M'*, many request identifier candidates are suddenly removed, then it is likely that *M* is a top-level method. Specifically, *lprof* counts the number of times each request identifier candidate appears in a log point in each method and accumulates this counter along the call-graph bottom-up. (See Figure 7 for an example.) Whenever this count *decreases* from method *M* to its caller *M'*, *lprof* concludes that *M* is a top-level method. The intuition is that developers naturally include identifiers in their log printing statements, and modifications to these identifiers are likely outside the top-level method.

In Figure 7, both `writeBlock()` and `readBlock()` accumulate a large count of request identifiers, which drops to zero in `run()`. Therefore, *lprof* infers `writeBlock()` and `readBlock()` are the top-level methods instead of `run()`. Note that although the count of `generationStamp` decreases when the analysis moves from `setGenerationStamp()` to `writeBlock()`, it does not conclude `setGenerationStamp()` is a top-level method because the accumulated count of all request identifiers is still increasing from `setGenerationStamp()` to `writeBlock()`.

4.3 Partial Order Among Log Points

In this step, *lprof* generates a Directed Acyclic Graph (DAG) for each top-level method (identified in the previous step) from the method's call graph and control-flow graph (CFG). This DAG contains each log point reachable from the top-level method and is used to help attribute log messages to top-level methods.

It is not possible to statically infer the precise order in which instructions will execute. Therefore, *lprof* takes the liberty of applying a number of simplifications:

1. Only nodes that contain log printing statements are represented in the DAG.

2. All nodes involved in a strongly connected component (e.g., caused by loops) are folded into one node. This implies that multiple log points may be assigned to a single node in the DAG.

3. Similarly, if there is a strongly connected component due to recursive calls, then those nodes are also folded into one.

4. Unchecked exceptions are ignored, since they will terminate the execution. Checked exceptions are captured by the CFG and are included in the DAG.

```
LOG.info("Request starts"); // log1
while (not_finished){
  r.process(); // log2 and log 3
}
LOG.info("Request ends"); // log4
```


Figure 8: DAG representation of log points.

As an example, Figure 8 shows the DAG generated from a code snippet. The asterisk (*) next to log 2 and log 3 indicates that these log points may appear 0 or more times. We do not maintain an ordering of the log points for nodes with multiple log points.

In practice, we found the DAG particularly useful in capturing the starting and ending log points of a request — it is a common practice for developers to print a message at the beginning of each request and/or right before the request terminates.

4.4 Thread Communication

In this step, *lprof* infers how threads communicate with one another. The output of this analysis is a tuple for each *communication pair*: (top-level method 1, top-level method 2, communication type, set of request identifier pairs), where one end of the communication is reachable from top-level method 1 and the other end is reachable from top-level method 2. "Communication type" is one of *local*, *RPC*, or *socket*, where "local" is used when two threads running in the same process communicate. A "request identifier pair" captures the transfer of request identifier values from the source to the destination; the pair identifies the variables containing the data values at source and destination.

Threads from the same process: *lprof* detects two types of local thread communications: (i) thread creation and (ii) shared memory reads and writes. Detecting thread creation is straightforward because Java has a well defined thread creation mechanism. If an instruction `r.start()` is reachable from a top-level method, where `r` is an object of class `C` that extends the `Thread` class or implements the `Runnable` interface, and `C.run()` is another top-level method, then *lprof* has identified a communication pair. *lprof* also infers the data-flow of request identifiers, as they are mostly passed through the constructor of the target thread object. In addition to explicit

thread creation, if two instructions reachable from two top-level methods (i) access a shared object, and (ii) one of them reads and the other writes to the shared object, then a communication pair is identified.

As an example, consider the HDFS code in Figure 3. *lprof* generates the following tuple: (writeBlock, PacketResponder.run, local, <DataXceiver.block.poolID, PacketResponder.block.poolID>, ..), indicating that `writeBlock()` could communicate with `PacketResponder` via local thread creation, and `poolID` is the request identifier used on both ends for the data value passed between the threads.

Threads communicating across the network: Pairing threads that communicate via the network is more challenging. While Java provides standard socket read and write APIs for network communication, if we naïvely pair the read to the write on the same socket, we would effectively end up connecting most of the top-level methods together even though they do not communicate. Consider the HDFS example shown in Figure 3. While `readBlock()` and `writeBlock()` do not communicate with each other, they share the same underlying socket.

Instead of pairing socket read and write, we observe that the sender and receiver that actually communicate both have to agree on the same protocol. Specifically, whenever *lprof* finds a pair of invoke instructions whose target methods are the serialization and deserializaition methods from the same class, respectively, the top-level methods containing these two instructions are paired. Developers often use third-party data-serialization libraries, such as Google Protocol Buffers [15]. This further eases *lprof*'s analysis since they provide standardized serialization/deserialization APIs. Among the systems we evaluated, Cassandra is the only one that does not use Google Protocol Buffers, but implements its own serialization library. For Cassandra, a simple annotation to pair `C.serialize()` with `C.deserialize()` for any class `C` is sufficient to correctly pair all of the communicating top-level methods. *lprof* also parses the Google Protocol Buffer's protocol annotation file to identify the RPC pairs, where each RPC is explicitly declared.

Improvements: To improve the accuracy of "log stitching", we add two refinements when pairing communication points. First, even when a thread does not contain any log point (which means it does not contain any top-level method), it will still be included in a communication pair if it communicates with a top-level method. In this case, its `run()` method will be used as the communication end point. The reason is that such a thread could serve as a link connecting two communicating top-level methods A and B. Not including the communication pair would prevent *lprof* from grouping the log messages from A and B.

```
Top-level  (DataXceiver.writeBlock, DAG#1, [id1,id2..])
methods:   (DataXceiver.readBlock, DAG#2, [id1,id2..])
.......................................................
DAGs:   entry -> log1 ---------> log4 -> exit
                    \            /
                   [log2, log3]*
.......................................................
Regex:  <log1, "Receiving block (.*):blk_(\d+)_\d*",
          id1:block.poolID, id2:block.block.blockID>
.......................................................
Comm.   (writeBlock, PktRsp.run, local, <id1,id1>, ..)
pairs:  (writeBlock, writeBlock, socket, <id1,id1>,..)
```

Figure 9: Output of *lprof*'s static analysis.

The second improvement is to infer the number of times a top-level method can occur in a communication pair. For example, a communication pair "(M1, M2*, local, ..)", where M2 is followed by an asterisk, means that method M1 could communicate with multiple instances of method M2 in the same request. The log analysis uses this property to further decide whether it can stitch messages from multiple instances of M2 into the same request. The inference of such a property is straightforward: if the communication point to M2 is within a loop in M1's CFG, then M2 could occur multiple times.

4.5 Summary of Static Analysis

The output of *lprof*'s static analysis is a file that contains the log printing behavior of the system. Figure 9 shows a snippet of the output file for HDFS. It consists of the following four segments:

1. *Top-level methods:* a list of tuples with (i) the name of the top-level method, (ii) an index into the DAG representation of the log points, and (iii) a list of request identifiers;
2. *DAGs:* the DAG for each top-level method;
3. *Log point regex:* the regular expressions for each log point and the identifier for each wildcard;
4. *Communication pairs:* a list of tuples that identify the communication points along with the identifiers for the data being communicated.

To speedup log analysis, this output file also contains a number of indexes, including: (i) an index of regular expressions (to speedup the matching of each log message to its log point) and (ii) an index mapping log points to top-level methods. This output file is sent to every machine in the cluster whose log is analyzed.

5 Log Analysis

The log analysis phase is implemented as a MapReduce job to group together information from *all* the log messages printed by each request. The map and reduce functions use a common data structure, called a *request accumulator* (RA), for gathering information related to the

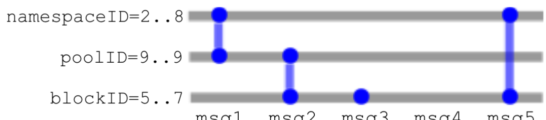


Figure 10: The grouping of five log messages where four print a subset of request identifier values.

same request. Each RA contains: (i) a vector of top-level methods that are grouped into this RA; (ii) the value of each request identifier; (iii) a vector of log point sequences, where each sequence comes from one top-level method; (iv) a list of nodes traversed, with the earliest and latest timestamp. The map and reduce functions will iteratively accumulate the information of log messages from the same request into the RAs. In the end, there will be one RA per request that contains the information summarized from all its log messages.

Map: Intra-thread Grouping

The map function is run on each node to process local log files. There is one map task per node, and all the map tasks run in parallel. Each map function scans the log file linearly. Each log message is parsed to identify its log point and the values of the request identifiers using regular expression matching. We also heuristically parse the timestamp associated with each message.

A parsed log message is added to an existing RA entry if and only if: (i) their top-level methods match, (ii) the identifier values do not conflict, and (iii) the log point matches the temporal sequence in the control flow as represented by the DAG. A new RA is created (and appropriately initialized) if the log message cannot be added to an existing RA. Therefore, each RA output by the map function contains exactly one top-level method.

Note that a sequence of log messages can be added to the same RA even when each contains the values of a different subset of request identifiers. Figure 10 shows an example. The 5 log messages in this figure can all be grouped into a same RA entry even though 4 of them contain the values of a subset of the request identifiers, and one does not contain the value of any request identifier but is captured using the DAG.

Combine and Reduce: Inter-thread Grouping

The combine function performs the same operation as the reduce function, but does so locally first. It combines two RAs into one if there exists a communication pair between the two top-level methods in these two RAs, *and* the request identifier values do not conflict. Moreover, as a heuristic, we do not merge RAs if the difference between their timestamps is larger than a user-configurable threshold. Such a heuristic is necessary because two RAs could have the same top-level methods and request identifies, but represent the processing of different requests (i.e., two writeBlock operations on the same block). This

```
node 3: 172.13.9.12
node 2: 172.13.9.28
node 1: 172.13.9.26
05:32:45,103 Receiving block 9..9:blk_5..7_1032
05:32:45,115 Received block 9..9:blk_5..7_1032
05:32:47,826 PacketResponder 9..9:blk_5..7_1032 terminating
Request accumulators after map
req. acc.1: ([writeBlock], {<id1:"9..9">, <id2:"5..7">},
 [[LP1]], [<172.31.9.26:"05:32:45,103 - N/A">])
req. acc.2: ([PktRsp.run], {<id1:"9..9">, <id2:"5..7">},
 [[LP2,LP3]], [<172.31.9.26:"05:32:45,115 - 05:32:47,826">])
Request accumulators after combine
req.acc.5:([writeBlock, PktRsp.run], {<id1: "9..9">, <id2:"5..7">},
 [[LP1],[LP2,LP3]], [<172.31.9.28:"05:32:45,847 - 05:32:47,567">])
req.acc.4: ([writeBlock, PktRsp.run], {<id1:"9..9">, <id2:"5..7">},
 [[LP1],[LP2,LP3]], [<172.31.9.28:"05:32:45,847 - 05:32:47,567">])
req.acc.3: ({writeBlock,PktRsp.run}, {<id1:"9..9">, <id2:"5..7">},
 [[LP1],[LP2,LP3]], [<172.31.9.26:"05:32:45,103 - 05:32:47,826">])
Request accumulators after reduce
req.acc.6: ({writeBlock,writeBlock,writeBlock,PktRsp.run,
 PktRsp.run,PktRsp.run}, {<id1:"9..9">, <id2:"5..7">},
 [[LP1],[LP1],[LP1],[LP2,LP3],[LP2,LP3],[LP2,LP3]],
 [<172.31.9.26:"05:32:45,103 - 05:32:47,826">,
  <172.31.9.28:"05:32:45,847 - 05:32:47,567">,
  <172.31.9.12:"05:32:46,680 - 05:32:47,130">])
```

Figure 11: The RAs that combine 9 log messages from 6 threads on 3 nodes belonging to a single write request in HDFS.

value is currently set to one minute, but should be adjusted depending on the networking environment. In an unstable network environment with frequent congestion this threshold should have a larger value.

After the combine function, *lprof* needs to assign a shuffle key to each RA, and all the RAs with the same shuffle key must be sent to the same reducer node over the network. Therefore the same shuffle key should be assigned to all of the RAs that need to be grouped together. We do this by considering communication pairs. At the end of the static analysis, if there is a communication pair connecting two top-level methods A and B, A and B are jointed together into a connected component (CC). We iteratively merge more top-level methods into this CC as long as they communicate with any of the top-level methods in this CC. In the end, all of the top-level methods in a CC could communicate, and their RAs are assigned with the same shuffle key.

However, this approach could lead to the assignment of only a small number of shuffle keys and thus a poor distribution in practice. Hence, we further implement two improvements to the shuffling process. First, if all of the communicating top-level methods have common request identifiers, the identifier values will be used to further differentiate shuffle keys.[4] Secondly, if an RA cannot possibly communicate with any other RA through network communication, we do not further shuffle it, but instead we directly output the RA into the request database.

Finally, the reduce function applies the same method

[4]Note that if a request identifier is not shared by all of the communicating top-level method, it cannot be used in the shuffle key because different communicating RAs might have different request identifier (e.g., one RA only has poolID while the other RA has blockID).

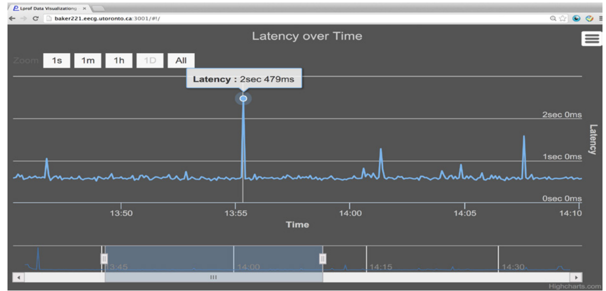


Figure 12: The web application that visualizes a request' latencies over time.

as the combine function. Figure 11 provides an example that shows how the RAs of log messages in the HDFS writeBlock request are grouped together. After the map function generates *req.acc.1* and *2* on node 1, the combine function groups them into *req.acc.3*, because `writeBlock()` and `PacketResponder.run()` belong to the same communication pair, and their request identifier values match. Node 2 and node 3 run the map and combine functions in parallel, and generate *req.acc.4* and *5*. *lprof* assigns the same shuffle key to *req.acc.3*, *req.acc.4*, and *req.acc.5*. The reduce function further groups them into a final RA *req.acc.6*.

Request Database and Visualization

Information from each RA generated by the reduce function is stored into a database table. The database schema is shown in Figure 1. It contains the following fields: (i) *request type*, which is simply the top-level method with the earliest time stamp; (ii) *starting and ending time stamps*, which are the MAX and MIN in all the timestamps of each node; (iii) *nodes traversed and the time stamps on each node*, which are taken directly from the RA; (iv) *log sequence ID (LID)*, which is a hash value of the log sequence vector field in the RA. For example, as shown in Figure 11, the vector of the log sequence of a writeBlock request is "[[LP1],[LP1],[LP1],[LP2,LP3],[LP2,LP3],[LP2,LP3]]". In this vector, each element is a log sequence from a top-level method (e.g., "[LP1]" is from top-level method `writeBlock()` and "[LP2,LP3]" is from `PacketResponder.run()`). Note the LID captures the unique type and number of log messages, their order *within a thread*, as well as the number of threads. However, it does not preserve the timing order between threads. Therefore, in practice, there are not many unique log sequences; for example, in HDFS there are only 220 unique log sequences on 200 EC2 nodes running a variety of jobs for 24 hours. We also generate a separate table that maps each log sequence ID to the sequence of log points to enable source-level debugging. We use MongoDB [28] for our current prototype.

We built a web application to visualize *lprof*'s analysis result using the Highcharts [21] JavaScript charting library. We automatically visualize (i) requests' latency over time; (ii) requests' counts and their trend over time; and (iii) average latency per node. Figure 12 shows our latency-over-time visualization.

System	LOC	workload	# of msg.
HDFS-2.0.2	142K	HiBench	1,760,926
Yarn-2.0.2	101K	HiBench	79,840,856
Cassandra-2.1.0	210K	YCSB	394,492
HBase-0.94.18	302K	YCSB	695,006

Table 1: The systems and workload we used in our evaluation, along with the number of log messages generated.

One challenge we encountered is that the number of requests is too large when visualizing their latencies. Therefore, when the number of requests in the query result is greater than a threshold, we perform down-sampling and return a smaller number of requests. We used the largest triangle sampling algorithm [39], which first divides the entire time-series data into small slices, and in each slice it samples the three points that cover the largest area. To further hide the sampling latency, we pre-sample all the requests into different resolutions. Whenever the server receives a user query, it examines each pre-sampled resolution in parallel, and returns the highest resolution whose number of data points is below the threshold.

6 Evaluation

We answer four questions in evaluating *lprof*: (i) How much information can our static analysis extract from the target systems' bytecode? (ii) How accurate is *lprof* in attributing log messages to requests? (iii) How effective is *lprof* in debugging real-world performance anomalies? (iv) How fast is *lprof*'s log analysis?

We evaluated *lprof* on four, off-the-shelf distributed systems: HDFS, Yarn, Cassandra, and HBase. We ran workloads on each system on a 200 EC2 node cluster for over 24 hours with the default logging verbosity level. Default verbosity is used to evaluate *lprof* in settings closest to the real-world. HDFS, Cassandra, and YARN use INFO as the default verbosity, and HBase uses DEBUG. A timestamp is attached to each message using the default configuration in all of these systems.

For HDFS and Yarn, we used HiBench [22] to run a variety of MapReduce jobs, including both real-world applications (e.g., indexing, pagerank, classification and clustering) and synthetic applications (e.g., wordcount, sort, terasort). Together they processed 2.7 TB of data. For Cassandra and HBase, we used the YCSB [11] benchmark. In total, the four systems produced over 82 million log messages (See Table 1).

System	Threads		Top-lev.	Log points	
	tot.	≥ 1 log*	meth.	≥ 1 id.	per DAG*
HDFS	44	95%	167	79%	8
Yarn	45	73%	79	66%	21
Cass.	92	74%	74	45%	21
HBase	85	80%	193	74%	30
Average	67	81%	129	66%	20

Table 2: Static analysis result. *: in these two columns we only count the log points that are under the default verbosity level and not printed in exception handler — indicating they are printed by default under normal conditions.

System	Correct	Incomplete	Incorrect	Failed
HDFS	97.0%	0.1%	0.3%	2.6%
Yarn	79.6%	19.2%	0.0%	1.2%
Cassandra	85.7%	9.6%	0.3%	4.4%
HBase	90.6%	2.5%	3.5%	3.4%
Average	**88.2%**	7.9%	1.0%	2.9%

Table 3: The accuracy of attributing log messages to requests.

6.1 Static Analysis Results

Table 2 shows the results of *lprof*'s static analysis. On average, 81% of the statically inferred threads contain at least one log point that would print under normal conditions, and there are an average of 20 such log points reachable from the top-level methods inferred from the threads that contain at least one log point. This suggests that logging is prevalent. In addition, 66% of the log points contain at least one request identifier, which can be used to separate log messages from different requests. This also suggests that *lprof* has to rely on the generated DAG to group the remaining 34% log points. *lprof*'s static analysis takes less than 2 minutes to run and 868 MB of memory for each system.

6.2 Request Attribution Accuracy

With 82 million log messages, we obviously could not manually verify whether *lprof* correctly attributed each log message to the right request. Instead, we manually verified each of the *log sequence IDs* (LID) generated by *lprof*. Recall from Section 5 that the LID captures the number and the type of the log points of a request, and the partial orders of those within each thread (but it ignores the thread orders, identifier values, and nodes' IPs). Only 784 different LIDs are extracted out of a total of 62 million request instances. We manually examined the log points of each LID and the associated source code to understand its semantics. The manual examination took four authors one week of time.

Table 3 shows *lprof*'s request attribution accuracy. A log sequence A is considered correct if and only if (i) all its log points indeed belong to this request, and (ii) there is no other log sequence B that should have been merged with A. All of the log messages belonging to a correct log sequence are classified as "correct". If A and B should have been merged but were not then the messages in both A and B are classified as "incomplete". If a log message in A does not belong to A then all the messages in A are classified as "incorrect". The "failed" column counts the log messages that were not attributed to any request.

Overall, 88.2% of the log messages are attributed to the correct requests.

7.9% of the log messages are in the "incomplete" category. In particular, 19.2% of the messages in Yarn were mistakenly separated because of only 2 unique log points that print the messages in the following pattern: "Starting resource-monitoring for container_1398" and "Memory usage of container-id container_1398..". *lprof* failed to group them because the container ID was first passed into an array after the first log point and then read from the array when the second message was printed. *lprof*'s conservative data-flow analysis failed to track the complicated data-flow and inferred that the container ID was modified between the first and the second log points, thus attributing them into separate top-level methods. A similar programming pattern was also the cause of "incomplete" log messages for Cassandra, HBase, and HDFS.

1.0% of the log messages are attributed to the wrong requests, primarily because they do not have identifiers *and* they are output in a loop so that the DAG groups them all together. This could potentially be addressed with a more accurate path-sensitive static analysis.

2.9% of the log messages were not attributed to any request because they could not be parsed. We manually examined these messages and the source code, and found that in these cases, developers often use complicated data-flow and control-flow to construct a message. However, these messages are mostly generated in the start-up or shut-down phase of the systems and thus likely do not affect the quality of the performance analysis.

Inaccuracy in *lprof*'s request attribution could affect users as follows: since the "incomplete requests" are caused by two log sequences A and B that should have been merged but were not, *lprof* would over-count the number of requests. For the same reason, timing information separately obtained from A and B would be underestimations of the actual latency. The "incorrect requests" are the opposite; because they should have been split into separate requests, "incorrect requests" would cause *lprof* to under-count the number of requests yet overestimate the latencies. Note that administrators should quickly realize the "incorrect requests" because *lprof* provides the sequence of log messages along with their source code information. The information about the "failed" messages, however, will be lost.

Number of messages per request: Figure 13 shows the cumulative distribution function on the number of mes-

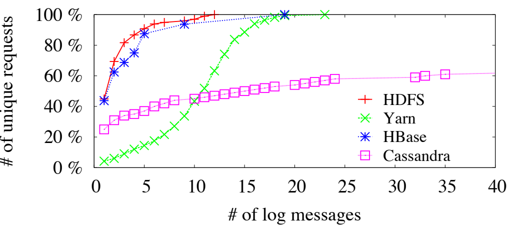


Figure 13: The cumulative distribution function on the number of log messages per unique request. For Cassandra, the number of nodes each streaming session traverses varies greatly, resulting in a large umber of unique log sequences (it eventually reaches 100% with 1484 log messages, which is not shown in the figure).

Category	example	tot.	helpful
Unnecessary operation	Redundant DNS lookups (should have been cached)	15	13 (87%)
Synch-ronization	Block scanner holding lock for too long, causing other threads to hang	4	1 (25%)
Unoptimized operation	Used a slow read method	2	0 (0%)
Unbalanced workload	A particular region server serves too many requests	1	1 (100%)
Resource leak	Secondary namenode leaks file descriptor	1	0 (0%)
Total	-	23	15 (65%)

Table 4: Evaluation of 23 real-world performance anomalies.

sages printed by each unique request, i.e., the one with the same log sequence ID. In each system, over 44% of the request types, when being processed, print more than one messages. Most of the requests printing only one message are system's internal maintenance operations.

6.3 Real-world Performance Anomalies

To evaluate whether *lprof* would be effective in debugging realistic anomalies, we randomly selected 23 user-reported real-world performance anomalies from the bugzilla databases associated with the systems we tested. This allows us to understand, via a small number of samples, what percentage of real-world performance bugs could benefit from *lprof*. For each bug, we carefully read the bug report, the discussions, and the related code and patch to understand it. We then reproduced each one to obtain the logs, and applied *lprof* to analyze its effectiveness. This is an extremely time-consuming process. The cases are summarized in Table 4. We classify *lprof* as helpful if the anomaly can clearly be detected through queries on *lprof*'s request database.

Overall, *lprof* is helpful in detecting and diagnosing 65% of the real-world failures we considered. Next, we discuss when and why *lprof* is useful or not-so-useful.

Table 5 shows the features of *lprof* that are helpful in

Analysis	helpful
Request clustering to identify bottleneck	73%
Log printing methods (inefficiencies are in the same method as the log point)	67%
Request latency analysis	33%
Per-node request count	7%

Table 5: The most useful analyses on real-world performance anomalies. The percentage is over the 15 anomalies where *lprof* is helpful. An anomaly may need more than one queries to detect and diagnose, so the sum is greater than 100%.

debugging real-world performance anomalies we considered. The "request count" analysis is useful in 73% of the cases. In these cases, the performance problems are caused by an unusually large number of requests, either external ones submitted by users or internal operations. For example, the second performance anomaly we discussed in Section 2 belongs to this category, where the number of verifyBlock operations is suspiciously large. In these cases, *lprof* can show the large request number and pinpoint the particular offending requests.

Another useful feature of *lprof* is its capability to associate a request's log sequence to the source code. This can significantly reduce developers' efforts in searching for the root cause. In particular, among the cases where *lprof* is helpful, 67% of the bugs that introduced inefficiencies were in the same method that contained one of the log points involved in the anomalous log sequence.

lprof's capability of analyzing the latency of requests is useful in identifying the particular request that is slow. The visualization of request latency is particularly useful in analyzing performance creep. For example, the anomaly to HDFS's write requests discussed in Section 2 can result in performance creep if not fixed. In addition, *lprof* can further separate the requests of the same type by their different LIDs which corresponds to different execution paths. For example, in an HBase performance anomaly [19], there was a significant slow-down in 1% of the read requests because they triggered a buggy code path. *lprof* can separate these anomalous reads from other normal ones.

In practice, the user might not identify the root cause in her first attempt, but instead will have to go through a sequence of hypotheses validations. The variety of performance information that can be SQL-queried makes *lprof* a particularly useful debugging tool. For example, an HBase bug caused an unbalanced workload — a few region servers were serving the vast majority of the requests while others were idle [18]. The root cause is clearly visible if the administrator examines the number of requests per node. However, she will likely first notice the request being slow (via a request latency query), isolate particularly slow requests, before realize the root cause.

In the cases where *lprof* was not helpful, most (75%)

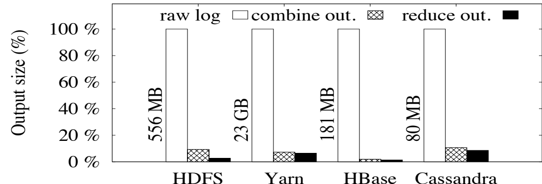


Figure 14: Output size after map, combine, and reduce compared to the raw log sizes. The raw log sizes are also shown.

were because the anomalous requests did not print any log messages. For example, a pair of unnecessary memory serialization and deserialization in Cassandra would not show up in the log. While theoretically one can add log messages to the start and end of these operations, in practice, this may not be realistic as the additional logging may introduce undesirable slowdown. For example, the serialization operation in Cassandra is an in-memory operation that is executed on every network communication, and adding log messages to it will likely introduce slowdown. In another case, the anomalous requests would only print one log message, so *lprof* cannot extract latency information by comparing differences between multiple timestamps. Finally, there was one case where the checksum verification in HBase was redundant because it was already verified by the underlying HDFS. Both verifications from HBase and HDFS were logged, but *lprof* cannot identify the redundancy because it does not correlate logs across different applications.

If verbose logging had been enabled, *lprof* would have been able to detect an additional 8.6% of the real-world performance anomalies that we considered since the offending requests print log messages under the most verbose level. However, enabling verbose logging will likely introduce significant performance overhead.

6.4 Time and Space Evaluation

The map and combine functions ran on each EC2 node, and the reduce function ran on a single server with 24 2.2GHz Intel Xeon cores and 32 GB of RAM.

Figure 14 shows the size of intermediate result. On average, after map and combine, the intermediate result size is only 7.3% of the size of the raw log. This is the size of data that has to be shuffled over the network for the reduce function. After reduce, the final output size is 4.8% of the size of the raw log.

Table 6 shows the time and memory used by *lprof*'s log analysis. *lprof*'s map and combine functions finish in less than 6 minutes for every system exception for Yarn, which takes 14 minutes. Over 80% of the time is spent on log parsing. We observe that when a message can match multiple regular expressions, it takes much more time than those that match uniquely. The memory footprint for map and combine is less than 3.3GB in all cases.

System	Time (s)		Memory (MB)	
	map+comb.	reduce	map+comb.	reduce
HDFS	14/528	21	185/348	1,901
Yarn	412/843	1131	1,802/3,264	7,195
Cassandra	4/9	17	90/134	833
HBase	3/7	2	74/150	242

Table 6: Log analysis time and memory footprint. For the parallel map and combine functions, numbers are shown in the form of median/max.

The reduce function takes no more than 21 seconds for HDFS, Cassandra, and HBase, but currently takes 19 minutes for Yarn. It also uses 7.2GB of memory. Currently, our MapReduce jobs are implemented in Python using Hadoop's streaming mode, which may be the source of the inefficiency. (Profiling Yarn's reduce function shows that over half of the time is spent in data structure initializations.) Note that we run the reduce job on a single node using a single thread. The reducer could and should be parallelized in real-world usage.

7 Limitations and Discussions

We outline the limitations of *lprof* through a series of questions. We also discuss how *lprof* could be extended under different scenarios.

(1) What are the logging practices that make lprof most effective? The output of *lprof*, and thus its usefulness, is only as good as the logs output by the system. In particular, the following properties will help *lprof* to be most effective: (i) attached timestamps from a reasonably synchronized clock; (ii) output messages in those requests that need profiling (multiple messages are needed to enable latency related analysis); (iii) the existence of a reasonably distinctive request identifier, and (iv) not printing the same message pattern in multiple program locations.

Note that these properties not only will help *lprof*, but also are useful for manual debugging. *lprof* naturally leverages such existing best-practices. Furthermore, *lprof*'s static analysis can be used to suggest how to improve logging. It identifies which threads do not contain any log printing statements. These are candidates for adding log printing statements. *lprof* can also infer the request identifiers for developers to log.

(2) Can lprof be extended to other programming languages? Our implementation relies on Java bytecode and hence is restricted to Java programs (or other languages that use Java bytecode, such as Scala). Similar analysis can be done on LLVM bytecode [24], but this would most likely require access to the C/C++ source code so it can be compiled to LLVM bytecode.

(3) How scalable is lprof? While the map phase is executed in parallel on each node that stores the raw log, the

reduce phase may not be evenly distributed. This is because all of the RAs that contain top-level methods that might communicate with each other need to be shuffled to the same reducer. This can result in unbalanced load. For example, in Yarn, 75% of the log messages are printed by one log point during the heartbeat process, and their RAs have to be shuffled to the same reducer node. This node becomes the bottleneck even if there are other idle reducer nodes. How to further balance the workload is part of future work.

(4) How does lprof change if a unique per-request ID exist? If such an ID exists in every log message, then there would be no need to infer the request identifier. The log string format parsing could also be simplified since now our log parser only needs to match a message to a log printing statement, but does not need to precisely bind the values to variables. However, the other components are still needed. DAG and communication pairs are still needed to infer the order dependency between different log messages, especially if we want to perform per-thread performance debugging. The MapReduce log analysis is still needed. If such an ID exists, then the accuracy of *lprof* will increase significantly, and we can better distribute the workload in the reduce function by using this ID as part of the shuffle key.

(5) What happens when the code changes? This requires *lprof* to perform static analysis on the new version. The new model produced by the static analysis should be sent to each node along with the new version of the system.

8 Related Work

Using machine learning for log analysis: Several tools apply machine learning on log files to detect anomalies [4, 29, 42]. Xu *et al.,* [42] also analyzes the log printing statements in the source code to parse the log. *lprof* is different and complementary to these techniques. First, these tools target anomaly detection and do not identify request flows as *lprof* does. Analyzing request flows is useful for numerous applications, including profiling, and understanding system behavior. Moreover, the different goals lead to different techniques being used in our design. Finally, these machine learning techniques can be applied to *lprof*'s request database to detect anomalies on a per-request, instead of per-log-entry, basis.

Semi-automatic log analysis: SALSA [40] and Mochi [41] also identify request flows from logs produced by Hadoop. However, unlike *lprof*, their models are manually generated. By examining the code and logs of HDFS, they identify the key log messages that mark the start and the end of a request, and they identify request identifiers, such as block ID. In contrast, *lprof* automatically infers the order relationship between log printing statements and the request identifiers from the program, and thus is not specific to one particular system. The Mystery Machine [10] extracts per-request performance information from the log files of Facebook's production systems, and it can correlate log messages across different layers in the software stack. To do this, they attach unique request identifiers to each log message. Commercial tools like VMWare LogInsight [26] and Splunk [38] index the logs, but requires administrators to do keyword-based searches on the log data.

Single thread log analysis: SherLog [43] analyzes the source code and a sequence of error messages to reconstruct the partial execution paths that print the log sequence. Since it is designed to debug functional bugs in single-threaded execution, it uses precise but heavyweight static analysis to infer the precise execution path. In contrast, *lprof* extracts less-precise information for each request, but it analyzes all the log outputs from all the requests of the entire distributed system.

Instrumentation-based profiling: Instrumentation-based profilers have been widely used for performance debugging [6, 8, 16, 17, 23, 32, 34]. Many, including Project 5 [1], MagPie [3], X-Trace [14], and Dapper [36], just to name a few, are capable of analyzing request flows by instrumenting network communication, and they can profile the entire software stack instead of just a single layer of service. G^2 [17] further models all the events into an execution graph that can be analyzed using LINQ queries and user-provided programs. In comparison, *lprof* is non-intrusive. It also provides source-level profiling information. However, it cannot provide any information if requests do not output log messages.

9 Conclusions

This paper presented *lprof*, which is, to the best of our knowledge, the first non-intrusive request flow profiler for distributed services. *lprof* is able to stitch together the dispersed and intertwined log messages and associate them to specific requests based on the information from off-line static analysis on the system's code. Our evaluation shows that *lprof* can accurately attribute 88% of the log messages from widely-used, production-quality distributed systems, and is helpful in debugging 65% of the sampled real-world performance anomalies.

Acknowledgements

We greatly appreciate the anonymous reviewers and our shepherd, Ed Nightingale, for their insightful feedback. This research is supported by NSERC Discovery grant, NetApp Faculty Fellowship, and Connaught New Researcher Award.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 74–89, 2003.

[2] Amazon found every 100ms of latency cost them 1% in sales. `http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/`.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation*, OSDI'04, 2004.

[4] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM Symposium on Foundations of Software Engineering*, FSE '11, pages 267–277, 2011.

[5] Boundary: Modern IT operation management. `http://boundary.com/blog/2012/11/19/know-your-iaas-boundary-identifies-performance-lags-introduced-by-cloud/`.

[6] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, EuroSys '07, pages 17–30, 2007.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI'06, pages 205–218, 2006.

[8] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, 2002.

[9] Chord: A program analysis platform for java. `http://pag.gatech.edu/chord`.

[10] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th symposium on Operating Systems Design and Implementation*, OSDI'14, 2014.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, OSDI'04, 2004.

[13] Moving an elephant: Large scale hadoop data migration at facebook. `https://www.facebook.com/notes/paul-yang/moving-an-elephant-large-scale-hadoop-data-migration-at-facebook/10150246275318920`.

[14] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: a pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, 2007.

[15] Google protocol buffers. `https://developers.google.com/protocol-buffers/`.

[16] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, 1982.

[17] Z. Guo, D. Zhou, H. Lin, M. Yang, F. Long, C. Deng, C. Liu, and L. Zhou. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'11, 2011.

[18] HBase bug 2399. `https://issues.apache.org/jira/browse/HBASE-2399`.

[19] HBase bug 3654. `https://issues.apache.org/jira/browse/HBASE-3654`.

[20] HDFS performance regression on write requests. `https://issues.apache.org/jira/browse/HDFS-4049`.

[21] Highcharts: interactive JavaScript charts for your webpage. `http://www.highcharts.com/`.

[22] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *26th International Conference on Data Engineering Workshops (ICDEW)*, pages 41–51, 2010.

[23] E. Koskinen and J. Jannotti. Borderpatrol: Isolating events for black-box tracing. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 191–203, 2008.

[24] The LLVM compiler infrastructure. `http://llvm.org/`.

[25] log4j: Apache log4j, a logging library for Java. `http://logging.apache.org/log4j/2.x/`.

[26] VMware vCenter Log Insight: Log management and analytics. `http://www.vmware.com/ca/en/products/vcenter-log-insight`.

[27] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ mapreduce for log processing. In *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011.

[28] Mongodb. `http://www.mongodb.org/`.

[29] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[30] Nagios: the industry standard in IT infrastructure monitoring. http://www.nagios.org/.

[31] NewRelic: Application performance management and monitoring. http://newrelic.com/.

[32] OProf - A system profiler for Linux. http://oprofile.sourceforge.net/.

[33] OpsView - enterprise IT monitoring for networks. http://www.opsview.com/.

[34] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation*, NSDI'06, 2006.

[35] M. Sharir and A. Pnueli. Two approaches to interprocedural analysis. *Program Flow Analysis, Theory and applications*, 1981.

[36] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[37] Simple logging facade for Java (SLF4J). http://www.slf4j.org/.

[38] Splunk log management. http://www.splunk.com/view/log-management/SP-CAAAC6F.

[39] S. Steinarsson. Downsampling time series for visual representation. *M.Sc thesis. Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland*, 2013.

[40] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: Analyzing logs as state machines. In *Proceedings of the 1st USENIX Conference on Analysis of System Logs*, WASL'08, 2008.

[41] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Mochi: Visual log-analysis based tools for debugging hadoop. In *Proceedings of the Conference on Hot Topics in Cloud Computing*, HotCloud'09, 2009.

[42] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of the ACM 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, 2009.

[43] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 143–154, 2010.

[44] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation*, OSDI'12, pages 293–306, 2012.

[45] D. Yuan, S. Park, and Y. Zhou. Characterising logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, 2012.

[46] Zabbix - an enterprise-class open source monitoring solution. http://www.zabbix.com/.

Pydron: semi-automatic parallelization for multi-core and the cloud

Stefan C. Müller[1,3], Gustavo Alonso[1], Adam Amara[2], and André Csillaghy[3]

[1]Department of Computer Science, ETH Zürich
[2]Department of Physics, ETH Zürich
[3]University of Applied Sciences Northwestern Switzerland

Abstract

The cloud, rack-scale computing, and multi-core are the basis for today's computing platforms. Their intrinsic parallelism is a challenge for programmers, specially in areas lacking the necessary economies of scale in application/code reuse because of the small number of potential users and frequently changing code and data. In this paper, based on an on-going collaboration with several projects in astrophysics, we present Pydron, a system to parallelize and execute sequential Python code on a cloud, cluster, or multi-core infrastructure. While focused on scientific applications, the solution we propose is general and provides a competitive alternative to moving the development effort to application specific platforms. Pydron uses semi-automatic parallelization and can parallelize with an API of only two decorators. Pydron also supports the scheduling and run-time management of the parallel code, regardless of the target platform. First experiences with real astrophysics data pipelines indicate Pydron significantly simplifies development without sacrificing the performance gains of parallelism at the machine or cluster level.

1 Introduction

In astronomy and other big-data sciences, the data generated by experiments and simulations is growing by leaps and bounds. Scientists have to use sophisticated computing infrastructures to be able to analyze and process all their observations.

Scientific data is often of a different nature than business data. Data from instruments and simulations has to be heavily processed before conclusions can be drawn from it. This process is repeated many times to calibrate and clean the data and to tune parameters and algorithms. Often this process is exploratory, using non-standard tools and ad-hoc developed code.

For example, in [29] Refregier et al. describe a procedure where repeated executions of a wide-field astronomy image simulator [3] are used to develop and calibrate the simulator, match the simulations with data from real observations, and perform a robustness analysis on the parameter space. With long lasting missions, such as the RHESSI spacecraft, launched in 2002 and still producing data today, changes to processing algorithms, data, and infrastructure happen continuously and will continue throughout the life time of the spacecraft and beyond [31]. This implies a constant correction of the data and the algorithms that adds significant overhead.

One can argue that today there are enough platforms – hardware and software – to support such application scenarios. However, this is far from being the case. The way to achieve performance today is through large scale parallelism: multi-core, rack-scale, or cloud computing. Current approaches to make parallelism available to developers typically provide either low level interfaces to parallel hardware (such as pthreads [7] or MPI [14] which are non trivial to use) or they require a complete integration into frameworks such as Spark [38] or Hadoop [13]. With fast changing code, legacy applications as well as legacy data formats, it is often impractical to apply such frameworks because of their rigid requirements in terms of data formats and algorithmic structure. This is not a question of the adequacy of these systems to the task at hand. It is a question of the total cost of adapting such a framework for the entire life cycle of a scientific mission. Code is often specific to an instrument and to the research of a single group. As a result, the economies of scale that would justify larger development efforts, as required to adopt existing frameworks, are just not there.

In this paper we argue for an alternative approach to separate application specific code from the parallelization framework (language and run-time). Our approach tries to provide maximum flexibility and ease of use for the application developer, with a system that takes care of parallelization, deployment, and scheduling on a variety of target infrastructures.

Our system, Pydron, can semi-automatically parallelize sequential Python code and run the result on multi-core, cluster, or cloud systems. The API consists only of two Python decorators: one to mark functions that should be parallelized, and one to mark functions that are free of side-effects. We use Python because of its wide spread use in the astronomy community. Pydron might even open the door for scientists to start using cloud based systems such as Hadoop and Spark by not requiring them to change their programming habits and not having to deal with the parallel infrastructure used to run the code.

Existing approaches, such as SEJITS [9], have automatically parallelized Python code when the code is restricted in form and data types, or introduced systems, such as CIEL [21], that can scale to multi-machine infrastructures when the application is written in a domain specific language. Pydron combines the advantages of both approaches.

With Pydron, the paper makes the following contributions:

- Pydron allows scientists to work in the language and with the tools they are familiar with, while giving them access to multi-core, cluster and cloud infrastructures.
- We show that the barrier for the application developer to benefit from modern infrastructures can be significantly lowered by using semi-automatic parallelization.
- We demonstrate how a dynamic data-flow graph can be used to counter the limitations of statical analysis when applied to a dynamic language.
- We present a system with three interchangeable elements that can be used to apply the ideas both to other languages and to other execution platforms – both hardware and software.

2 Related Work

Big data in scientific applications has lead to a large variety of systems to make the use of high performance infrastructures simpler for the developer.

Science Data Archives A significant effort has been made to simplify the analysis of scientific data once it has been collected and processed into science-ready products. For data that can be represented in tabular form, e.g. the Sloan Digital Sky Survey [33], databases are typically used. The large data volumes and the sophisticated queries can made specialized extensions to the database system necessary [32]. When data does not easily fit into a relational data model, other approaches are required, such as SciDB [5], a database system that generalizes the relational concept to multidimensional arrays to better support data types such as images or spectra. Many of these extensions are application specific and are rarely used in other contexts.

Delayed Execution Before data can be analyzed, it needs to be processed. Many of the languages currently in widespread use were not designed with parallelization in mind, which leads to a demand for easy-to-use interfaces between the language and the parallel infrastructure. One approach, used by Spark [38], Weaver [6], or FlumeJava [10] is to collect expressions during the execution of the program. Instead of directly executing an operation, an object is returned that represents the not-yet-calculated result. Those objects can then participate in other operations, resulting in an expression graph. The expression graph is then evaluated in parallel.

Such systems typically introduce a set of data types together with operations that can be applied to them. The close control of the system over both data and operators allows for efficient parallelization and sophisticated data management. However, it also forces the developer to formulate the code using only the data formats and operators provided. For the scientific applications we are interested in, this is a significant limitation as those applications often use legacy code and data formats.

Source-to-Source Translation The approach used by SEJITS [9] or Parakeet [30] is to translate the source code into another programming language, such as CUDA [22] or C++, more suited for the targeted infrastructure. The translation and subsequent compilation typically happens just-in-time during execution. The performance improvement comes from a more efficient target language, and / or from parallel execution on hardware such as graphics processor units. These systems place constraints on the code they can translate: Not all data types and operations may be available in the target language, and since Python is dynamically typed, the systems typically require complete type inference. For example, both SEJITS and Parakeet operate only

on NumPy [17] data types and cannot handle other objects. This makes such systems attractive to speed up inner loops, where the amount of translated code is relatively small, and it it easier to comply with the constraints. Thus, parallelization is typically fine grained. Pydron targets infrastructures on which Python is available, and can therefore avoid the constraints that result from a translation into a different language.

Domain Specific Languages When parallelizing at a coarse granular level, near the outer most loops, the distributed tasks will use a significant amount of unmodified application's code. Applying strong constraints is less practical. SWIFT [36], CIEL [21], or PigLatin [24], use custom 'orchestration' languages in which the outermost loops are parallelized.

Parallelization of the actual computations in the inner loops are typically not addressed in those languages. Instead, such systems provide convenient ways to call code written in other languages. The orchestration languages are often functional or have other means to avoid side-effects which would hinder automatic parallelization. Those constraints are less restrictive than in the source-to-source translation approach since they only apply to the outer loops and not to the code called from within. Systems such as Taverna [23] also belong to this category. They use a graphical programming language, in the form of a work-flow graph, to specify the orchestration of the computation. A separate language enforces a strict separation between orchestration and computation. Pydron blurs the barrier between orchestration and computation and avoids the learning curve of an additional language.

Streaming Data streaming systems such as Spark Streaming [39] and Naiad [20] use a data-flow graph representation. Records are passed along the edges, an the vertices represent operations on them. Several of the non-streaming systems also use graph representations internally. In those systems vertices are executed once, and data that flows along the edges typically represent larger units of data (for example sets of records). Data streaming can achieve finer parallelization, on the granularity of individual records, while keeping the graph at a manageable size since there is no need to have a vertex per record.

For use-cases that can be formulated as record streams, such systems can scale well to many nodes. The developer has to provide the implementation of the vertices and, unlike Pydron, also has to provide the structure of the graph, for example with a SQL or LINQ [37] expression.

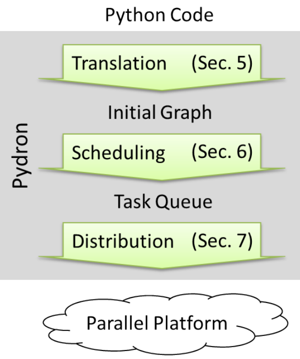


Figure 1: Pydron Overview

Static Compiler Optimization Modern processors support parallelism in various ways, with multiple cores, multiple threads, and vectorized instructions. Compile-time optimizations to make use of those features have been studied extensively [12]. Those approaches are limited by the large search space of such optimizations. As a result such optimizations are typically only applied to relatively simple innermost loops and not at higher levels.

In contrast to these systems, Pydron parallelizes regular Python code, similar to compiler optimizations or source-to-source translation, but it uses a coarser granularity and scales beyond shared memory systems. Existing systems that can scale to such an infrastructure use either domain specific languages or force the developer to formulate the problem in a form dictated by the system. In Pydron, however, the parallelization and execution are separated. As a result, Pydron can easily target either multi-core, clusters, or cloud platforms (or a combination there of). Our approach supports the complete Python language, without the constraints of existing source-to-source translation systems. Since we do not translate the distributed code into a different language, we are not limited to support only those data types and functions from Python libraries that have equivalents in the target language.

3 System Overview

Pydron operates by translating Python into an intermediate data-flow graph representation. The graph is then evaluated by a scheduler sub-system which uses a dis-

tribution sub-system to execute individual tasks to the worker-nodes.

Dynamic-typing, late-binding, and side-effects makes static analysis of Python code hard. We use a dynamic data-flow graph to account for the dynamic nature of Python. This is is a similar approach as used by CIEL [21], and we too use dynamic changes to the graph to handle data dependent control flow, such as loops and branches. We take this idea a step further and continuously refine the data-flow graph to incorporate information about object types and the data itself as it becomes known during execution.

Pydron consists of three components (Figure 1):

The *translator* transforms Python code of those functions decorated with @*schedule* into their initial graph representation. All language constructs can be translated. The translation happens at run-time, the first time the function is invoked. The translation process is described in Section 5.

When a translated function is invoked, the *scheduler* component analyzes the graph to decide in which order the tasks have to be executed and which of them may run in parallel. It fills a queue with tasks that are ready for execution. When the results become known after execution, the scheduler is informed. The scheduler is responsible for making the dynamic graph changes and to add those tasks that have now become ready for execution to the queue. Scheduling is described in Section 6.

The tasks in the queue are distributed to worker nodes for execution. The *distribution system* is responsible to acquire the resources and to start the Python interpreters (typically one per core) which will execute the tasks, as well as to release the resources at the end. It also deploys the the user's application on the worker nodes. We have several back-ends implemented to support cloud, cluster and multi-core infrastructure. The distribution system is described in Section 7.

Pydron has been designed to make these components interchangeable so as to allow extensions to target other languages and execution platforms. The components described in this paper focus on achieving full support for the Python language with greatest flexibility for the developer.

4 Language API: Decorators

To make the system as easy to use as possible, the API of Pydron consists only of two decorators. @*schedule* marks the functions which should be considered for automatic parallelization. This allows the developer to control which parts of the application the system will parallelize. Since the developer marks complete methods with @*schedule*, and not individual loops or statements, this is typically a simple task as most applications will have only a few central functions that orchestrate the process.

The @*functional* decorator informs Pydron that the marked function is free of side-effects and may be run on a different machine. The function has to meet the following criteria:

- No modification of objects passed as arguments.
- Arguments and return values need to support serialization with Python's pickle API [25].
- No assignments to global variables.
- No environment interactions.

The criteria apply only to the observed behavior of the function, not to every operation within its implementation. Especially, the last criterion can be interpreted rather freely as it isn't a technical constraint of Pydron.

Environment interactions are not tracked by our system and could lead to non-deterministic behaviour if executed in parallel. Sometimes this can be acceptable. If, for example, log messages are generated inside a function marked with @*functional*, a non-deterministic order of the log messages should be acceptable.

Another common situation is file IO. Open file handles cannot be passed to marked functions since Pydron has currently not support for remote file operations. Often it is sufficient to track the files by their filenames. If the function only reads files for which it has received the corresponding filename as an argument and returns the name of the files it has written, then the function can typically be safely marked as @*functional*. Pydron will track the file dependencies between the functions by tracking their names, enforcing the correct order of execution. This is especially handy when operating on clusters with a shared file system. In astronomy, many codes already use files to store intermediate data products, making this workaround particularly simple.

We don't currently automatically check if the conditions for the @*functional* decorator hold, even though some automatic checks could be implemented to support the developer in this decision.

5 Language Translation

The intermediate data-flow graph used by Pydron is directional, acyclic, and bipartite. There are two types of nodes: Value-nodes which represent immutable data and tasks which represent operations on data.

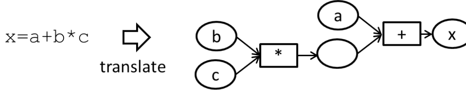


Figure 2: Operator Translation

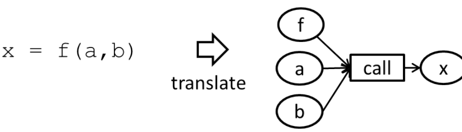


Figure 3: Function Call Translation

Figure 2 shows the data-flow graph for a simple expression. In general, variables become value-nodes. Expressions and statements become tasks. Intermediate values in expressions are also represented by value-nodes. Those have no direct correspondence with a Python variable, but behave no differently otherwise.

Functions with the @*schedule* decorator are translated the first time they are invoked. Pydron uses Python's built-in parser to create an abstract syntax tree of the function's code. This tree is then traversed twice. A first pass identifies the scope of the variables. The actual translation happens in the second pass.

In work-flow systems such as Taverna [23] there are also other type of edges. In Pydron dependencies between tasks can only result from data dependencies.

5.1 Function Calls

Functions are first-class objects in Python. The invoked function is represented by a value-node since the function may itself be the result of an operation. This value-node is an input to a *call*-task, together with the arguments passed to the function. The return value is again a value-node. Figure 3 shows a simple example.

Pydron supports all of Python's language features for function calls, such as keyword arguments and argument lists.

In general, we cannot know at translation time which function is invoked. We have to assume that it may have side-effects or modify arguments passed to it in-place. This leads to additional edges connected to the task (as described in Section 5.3).

To improve the readability of the data-flow graphs in this paper, we show a slightly compacted form. Instead of showing the function calls as in Figure 3, we hide the input for the function object and name the *call* task-node by the function. We will also hide intermediate value-nodes, as shown in Figure 2, from expressions. Instead

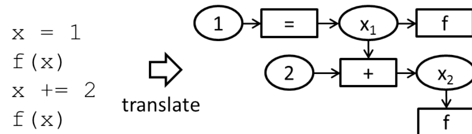


Figure 4: Static Single Assignment Form

we directly connect the two tasks.

5.2 Static Single Assignment Form

Python variables can be reassigned. This conflicts with the property that value-nodes represent immutable data. Therefore a one-to-one relationship between variables and value-nodes is not possible. We translate the Python code into a static single assignment form [11]. A Python variable is represented by a series of value-nodes, each representing the content the variable would hold for a period of the time in a sequential execution of the code.

Figure 4 shows an example. The variable x is assigned twice. The value node x_1 represents the content before the += operator is executed, x_2 represents the content after. Once this operator has been executed, both x_1 and x_2 are known and the scheduler (see Section 6) will be able to schedule both calls to f for parallel execution. We don't show the subscripts explicitly in the other figures as the order can be derived easily from the graph structure.

5.3 In-place modifications

Python objects can change after their creation. This poses a problem since value-nodes represent immutable data. Creating a copy before a modification is impractical since the data of the value-node may not have value semantics.

Pydron uses another solution based on the following observation: The value represented by a value-node becomes known once the producing task has been executed. The value becomes permanently unknown after an in-place modification on the value-node. Conceptually, the value-node still represents the unchanged value. If the task which performs the in-place modification is executed after all other tasks that use that value-node have completed, then the modification will not have an observable effect as the value-node will no longer be needed.

An operation with a potential in-place modification is translated differently. The input edge that connects the task with the affected value-node is flagged as *last-read*. A new value-node to represent the modified value is added as an output, in accordance with the static sin-

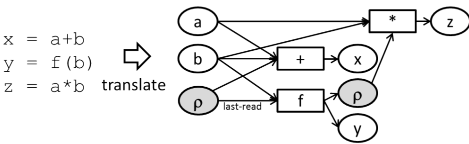


Figure 5: Synchronization point

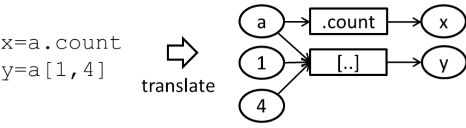


Figure 6: Attributes and Subscripts

gle assignment form (see Section 5.2). The scheduler is aware of the *last-read* flag and ensures correct ordering of task-executions.

In the general case, this is still insufficient to guarantee the same results since the objects changed in-place may be referenced from other value-nodes as well. For each affected value-node, the above special translation would have to be applied. We cannot identify them in general. Therefore, whenever there is a risk of having an in-place modification that may affect other value-nodes, we make the task into a synchronization point.

A pseudo variable is introduced to model synchronization points. This variable is implicitly read by all tasks. A synchronization point is translated as an in-place modification on this variable. The *last-read* edge created by that translation ensures that all previous tasks have to be executed before the synchronization point. All tasks after the synchronization point will use the new value-node, representing the changed pseudo variable, as an input, and will therefore be forced to wait til after the synchronization point.

Synchronization points are also used to model operations with potential side-effects. Such as a call to a function which is not marked with @*functional*. Since the invoked function is not known at translation time, all calls are initially translated as a synchronization point. We will use the adaptive graph refinement to remove those synchronization points for @functional functions.

Figure 5 shows an example. The *last-read* flag forces the addition to complete before f is invoked. The assignment of the pseudo-variable ρ, forces the multiplication to execute after f.

There would be more straight-forward ways to model synchronization points, for example by having ρ as an output of the addition, but this method allows us to reuse the technique of in-place modifications, reducing the overall complexity of the system.

5.4 Attribute and Subscript

When used as a right-hand-side expression, both attributes and subscripts are translated to a task which receives the object as an input. For attribute access, the name of the attribute is stored in the task. In case of a subscript, the indices are also passed as inputs. Pydron supports all indexing constructs, including slicing. Figure 6 shows a simple example with both attribute and subscript used as a right-hand-side expression.

Both attributes and subscripts can be used as a left-hand-side expression as well. Those tasks have the assigned value as an additional input. By its nature, such an assignment is an in-place modification on the object, for which additional edges have to be added to the graph, as described in Section 5.3.

5.5 @*functional* Decorator

Functions decorated with @*functional* are not changed at all. The only effect is that Pydron internally keeps track of those functions.

When the function object on a *call*-task (see Section 5.1) becomes known during the evaluation of the graph, the scheduler checks if the invoked function is @*functional*. If so, the graph is changed to remove the synchronization point.

This usually happens quite early during the evaluation of a graph as most invoked functions will be stored in global variables (Section 5.9) and are not the result of operations.

5.6 Conditional Statements

The translation of the *if* statement makes use of the dynamic data-flow graph. The complete *if* statement is initially translated into a single task. The condition is an input to this task. Both the *body* and the *else* section are translated individually into sub-graphs.

During translation of the *bod*y and *else* sections, the variables read and assigned are kept track of. They too become inputs and outputs of the *if*-task.

A variable in Python can have an undefined content if it is assigned in only one of the sections. In the data-flow graph each value-node must be the output of a task. For such situations a special task is added to the graph which produces an undefined value as a result. The scheduler is aware of value-nodes with an undefined content and will produce the same exceptions on an attempt to use the

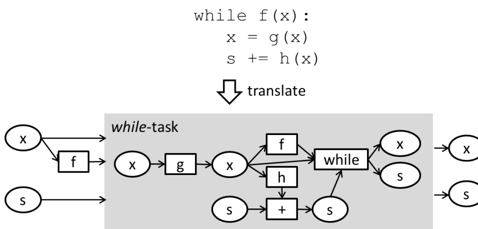


Figure 7: *while* Loop Translation

```
for i in items:
   f(i)
   if i == end:
     break
   g(i)
```

reformulation

```
for i in items until break_flag:
   f(i)
   if i == end:
     break_flag = True
   if not break_flag:
       g(i)
```

Figure 8: Reformulate *break* Statement

value-node as Python would when using an undefined variable.

5.7 Loops

The translation of *for* and *while* loops use the same techniques as for conditional statements. The loop *body* and the optional *else* section are translated into sub-graphs. The complete loop construct is translated initially into a single task. The condition, in case of a *while* loop, or the iterator, in case of a *for* loop, is an input to this task. The expression of a *for*-loop evaluates to an iterable. We insert another task which uses the built-in function *iter()* to get the iterator.

At the end of the body sub-graph the loop task itself is added to form a tail-recursive pattern. This may seem to enforce sequential execution, and indeed it will do so, unless the scheduler finds the requirements met at run-time that allow parallel execution the iterations.

Figure 7 shows an example. The *while* loop is first translated into a single *while*-task, internally storing the sub-graph of its *body*. For every variable read in the body there is a corresponding input, and for every variable assigned there is an output. The sub-graph also contains the inner *while*-task which forms the tail recursion.

5.8 *return*, *break*, and *continue*

The three statements *return*, *break*, and *continue* interrupt the regular control flow. Pydron translates those statements by reformulating them with conditional expressions and flag variables. Figure 8 shows an example. The interrupting statement sets a flag variable. Once such a flag has been set, all code afterwards is put into a condition checking the flag. Since the interrupting statement might be inside nested *if* statements, multiple conditions might be introduced. The task of the loop is aware of the flags and uses them to decide if to replace the task by the *body* sub-graph or if to end the loop, with or without a final replacement with the *else* sub-graph. In case of the return statement, the *return* value is stored together with the flag. This reformulation is performed on the fly during the translation into the data-flow graph.

5.9 Non-local variables

Besides being local to the function, variables can also belong to the module in which the function is defined (global variables). In addition Python allows functions to be defined within functions. Those nested functions may access the variables of the enclosing function (closure variables). Pydron supports both global variables, closures, and nested functions.

Python stores the value of a closure variable in a *cell*-object that lives on the heap. Any access to the variable is transparently transformed into an access to this *cell*-object. We can translate functions containing closure variables by using the same approach as Python: A *read-cell* or *write-cell* task is added to the graph whenever a closure variable is accessed. The task is reading or writing the *cell*-object when executed. This has to be done in both the enclosed and the enclosing function. Pydron identifies the variables that are accessed from enclosed functions in a first pass over the abstract syntax tree.

Any access to global variables can be translated similarly with *read-global* and *write-global* tasks. Assignment to a global variable is considered a side-effect and leads to a synchronization point. Reading a global variable does not.

5.10 Exception Handing

Exception handling statements such as *try-except* or the *with* statement are translated as well.

At first, exceptions seem to forbid any parallelism as every operation could potentially throw an exception. The decision if an operation is to be executed can only be made once the previous operation has finished. We

can still achieve parallelism by using speculative execution [27]. An operation is executed even if it is not clear if an exception in a previous operation may occur. This is possible because when we execute a task without side-effects or in-place modifications and discard its outputs then this has the same observable behavior as if we would not have executed the task at all.

If a task does have side effects, this translates into a synchronization point which forces all previous operations to complete before it. In this situation we know if any of the previous operations raised an exception.

The cost of this method is that we potentially waste significant resources on speculatively executed tasks should an exception occur. If we assume that exceptions are used in rare scenarios and not for regular control flow, then exception handing has little impact on the potential parallelism.

5.11 *yield* Statement

The *yield* statement is special since it transforms the function in which it appears into a generator. When the function is invoked, the execution of the function pauses and an iterator is returned. When elements are consumed from the iterator, the execution proceeds from *yield* to *yield* statement.

The *yield* statement can be translated into a data-flow equivalent which is treated specially by the scheduler. Between reaching a *yield* statement and the next consumption of an element on the iterator, only tasks can be executed which are free of side-effects and perform no in-place modifications. This is similar to exception handing as we cannot say for sure if another element will be consumed by the iterator, making the execution of any operation after a *yield* statement speculative.

6 Scheduling

The scheduler component of Pydron takes as input the graph produced in the translation step and produces as output a continuously updated list of tasks to be executed.

The scheduler becomes active when a function marked with @*schedule* is invoked. It keeps track of the execution progress and enforces the correct order of execution and decides which tasks may run in parallel.

A task is ready for execution if the following conditions are met:

- All its inputs are known.
- The task does not require further changes of the graph.
- For any input with the *last-read* flag (see Section 5.3) all other tasks that share this input have already completed.

This guarantees the correct order of execution. If multiple tasks fulfill those criteria, they may execute in parallel.

All tasks that are identified as ready for execution are placed in a queue. This queue is read by the distribution system (Section 7). The distribution system informs the scheduler once the execution of a task has completed. The outputs of the finished tasks become known, potentially leading to more tasks becoming ready for execution.

The scheduler also uses the information that becomes available during execution for refinement of the data-flow graph. The availability of run-time information allows for various optimizations, of which a small number has already been implemented in Pydron.

6.1 Adaptive Graph Refinement

Some tasks will require changes to the graph. Every time the value of a value-node becomes known the scheduler informs the tasks which have this value-node as an input and allows them to change the graph. There are two kinds of changes made to the graph:

- Removal of a synchronization point (Section 6.2).
- Replacement of the task-node by a sub-graph (Section 6.3).

6.2 Removal of synchronization points

The dynamic nature of Python often doesn't allow to make strong guarantees from the code alone. This forces us to translate the code into a graph with many synchronization points. The most common cause are *call* expressions, since the invoked function is not known at translation time (see section 5.3). Once the called function becomes known, the scheduler can check if it is marked as @*functional*. If so, the synchronization point is removed. The two value-nodes for ρ are merged into one and the *last-read* flag is removed.

In most codes, the functions themselves are not the result of expressions, but are either globally defined or object attributes, therefore most synchronization points can often be removed early in the execution.

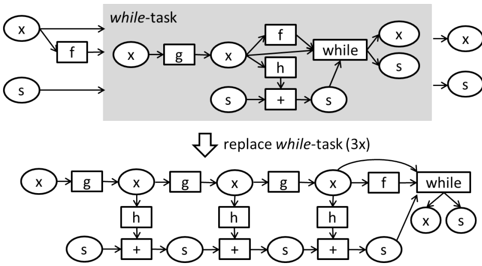


Figure 9: *while* loop parallelization

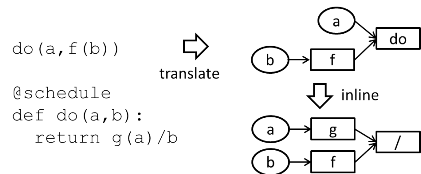


Figure 10: Inline Substitution

6.3 Loops

After the translation, there is a single node in the graph for a loop (section 5.7). Loop tasks are replaced by their sub-graphs during scheduling. According to the loop condition, the scheduler replaces the looop task with either its *body* or *else* sub-graph. Since the *body* sub-graph contains the loop node again, a tail-recurison is formed. At all stages the graph is acyclic which is a different approach as taken by Naiad [20] were loops are modeled with feed-back edges and the control flow is implemented with timestamps on the records passed through the graph.

If the decision for the execution of the next iteration depends on the complete execution of the body, or if the body contains a synchronization point, then the replacement of the tail-recursive tasks must happen after executing all previous tasks, resulting in a sequential execution. But if the condition is known early, as it is often the case with *for*-loops, then the complete loop can be unrolled in a short time.

Figure 9 shows the graph from the *while* loop of Figure 7 after three replacements. This method can still allow for parallelism even if the loop iterations are not completely independent of each other. If g executes faster than h, the *while* loop will unroll faster than a single h executes, allowing for several parallel executions of h. Even if g is slow, g can run in parallel with the h call of the previous iteration.

The summation is still executed sequentially, without making any associativity assumptions on the potentially overloaded plus operator.

6.4 Scheduler Relocation

The scheduler can run on the workstation of the user, but it can also be relocated to a remote Python process managed by the distribution system. If the latency between the user's workstation and the remote machines is substantial (such as when executing on a cloud), this will greatly reduce the communication overhead. The data transferred from and to the workstation is reduced to the arguments passed to the @*schedule* function, the data-flow graph of the function, and the return value.

6.5 Inline Substitution

If, during the evaluation of the graph, an invoked function is found to be decorated with @*schedule*, then this function can be translated to a data-flow graph as well.

Instead of invoking the original Python method, the *call*-task is replaced by the function's graph. This corresponds to the inline substitution optimization performed by compilers [11]. Inline substitution can expose additional parallelism as shown in Figure 10. The call to *do* is inline substituted, allowing for parallel execution of f and g, even though f is required to calculate an argument of the call. This works since the substitution can be performed as soon as the invoked method is known, even before the arguments are calculated.

Inline substitution is optional and the scheduler may decide not to inline a call and instead run the original, untranslated, function to control the granularity of the parallelization, depending on the target execution platform.

6.6 Scheduler-local Execution

Some tasks, notably those with side-effects, cannot be distributed safely. Such tasks are executed directly within the thread of the scheduler. Since they enforce a synchronization point, such tasks cannot be executed in parallel anyway.

For some tasks, it might not be worth the effort of sending them to a worker node for parallel execution even though it would be formally possible. For example, multiplying two integers has no side-effects, but the overhead of distributing this task is in no relation to the cost of the operation itself. The scheduler can decide to run such tasks locally.

Pydron currently applies a simple heuristic based on the type of the operation. Since the decision has to be

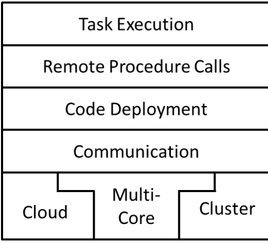


Figure 11: Layers of the distribution system

made only once all the input values to the task are known, quite elaborate techniques could be used, based on operation and data size information, to balance communication and execution cost.

7 Distribution

The distribution system takes tasks that are ready for execution and distributes them to Python run-times, potentially on another machine. It is able to automatically allocate and free resources, so that the user only has to run the application on a local workstation, as one would for regular Python. The system is highly configurable to support different infrastructures. Several configurations can be prepared for the user to choose from.

During operation, the system typically maintains several running Python processes on which it can execute tasks in parallel. The task, and all the inputs, are transferred to the Python process where the task is executed. The outputs are transferred back to the scheduler.

The main effort of the distribution system is to start or acquire the worker nodes, start the Python processes, and establish the means required for remote method invocation. The distribution system uses a layered architecture (Figure 11).

7.1 Worker-Node Acquisition

Before the Python processes can be started, the machines need to be acquired. There are several implementations of this layer. Each provides the means to acquire machines, run a command on them, and release the machines. This API is then used to start a Python process with a small boot-strap script passed to it as an argument.

Multi-core Back-end Python's global interpreter lock makes threads unusable for exploiting multi-core machines, we therefore use multiprocessing instead. This back-end starts processes on the local machine using the *subprocess* module provided by Python.

Cluster Back-end Instead of starting Python locally, a job is submitted to the cluster's job queue, asking for a number of nodes on which the process is run. The job submission is done with a configurable bash script. Pydron can also execute this bash script remotely via an SSH connection as it is often required for clusters with a login-node.

Cloud Back-end The cloud back-end first starts worker nodes. Pydron is using Apache libcloud [1] which supports various commercial cloud providers. Once the instances have started, Pydron opens an SSH connection to each to execute the command. The disk image which is booted can be configured, as can the type of the nodes. The image must contain a Python installation and allow SSH access. Neither Pydron nor the user's application need to be installed on it.

Combining Back-ends The multi-core back-end is often combined with the cluster or cloud back-end to make use of multiple cores on multiple machines.

7.2 Establishing Communication

The boot-strap script establishes communication. Pydron currently supports communication via TCP connections. To mitigate problems caused by firewalls and network address translation, connection attempts are made in both directions. Other methods, such as MPI [14] could also be implemented.

Each node has one communication channel to the workstation from which Pydron was first started. Additional channels for direct communication with other participating nodes are opened on demand, as is needed when the scheduler is relocated (see Section 6.4) to a worker node.

7.3 Code deployment

To execute tasks on remote nodes, the application's code has to be available on the nodes. Manual deployment of the code can be tedious, especially if the nodes do not have access to a shared file system.

Pydron automates this process by using a Python import hook [34] on the worker nodes. When a Python module is imported which is not available on the worker node, the import hook loads the source code from the user's workstation over the established communication channel. Python's internal caching of loaded modules ensures that this has to be done only once per module and Python process.

The code of Pydron itself is also transferred to the worker nodes. This simplifies the deployment of Pydron, as it does not need to be installed on every node. It also avoids potential version compatibility issues as all participants use exactly the same version of Pydron.

7.4 Third-party libraries

The code deployment system also works for most third-party libraries. This further simplifies the deployment of the code on the worker nodes, and makes Pydron more transparent to the user. The exception are libraries that contain native code. Pydron currently does not attempt to transmit native code libraries. In some cases, particularly when the worker-nodes are binary compatible, or when the libraries can be installed via Python package repository, automatic deployment of such libaries could be made possible, but this currently not implemented.

An example of a library which has to be manually deployed is SciPy [17]. This does not prevent us from using SciPy. The primary data type, NumPy arrays, are serializable with pickle. The methods of SciPy which do not change the data in-place can be whitelisted as *@functional*.

7.5 Remote Procedure Call

A simple remote procedure call (RPC) protocol is established on top of the communication channel. It uses Python's *pickle* API to serialize the invoked function, the arguments passed to it, as well as the return value or the raised exception.

7.6 Executing Task-Nodes

RPC connections are established from the node on which the scheduler is running to all other available nodes. The distribution system keeps track of idle and busy workers. Tasks added by the scheduler to the queue of ready tasks are assigned to idle workers. The task is then executed on the node using an RPC call. The result is passed back to the scheduler.

7.7 Fault Tolerance

With increased number of nodes, the probably of a single node failing is greatly increased. The distribution system is in charge of monitoring the Python processes. If a process fails to react, it is taken out of the set of available workers. If it was executing a task, this task is put back to the queue of tasks ready for execution. Since only tasks without side-effects are executed on remote nodes there are no conflicts arising from executing a task twice.

Currently, we follow a simple policy of rescheduling failed tasks. In the future we will explore more complex policies that could take user input into consideration.

8 Discussion

It is the simplicity of use and design that makes Pydron attractive for domains such as astronomy that lack the economy of scale to justify porting efforts to a different language or to other frameworks. The system works without sophisticated language analysis, scheduling, or resource management. Using more advanced implementations for those components will certainly improve the performance further. CIEL [21], in particular, is a system that contains many components from which Pydron could profit.

Pydron shares the architecture with systems such as CIEL and Dryad [16]: An orchestration language is translated into a data-flow graph. The individual tasks, represented by nodes, are typically written in a language such as Java. They are sent to worker nodes for execution. Such systems have the advantage over approaches such as MapReduce [13] in handling iterative computations [21]. In Section 2 we discussed how a separate orchestration language can be a barrier to adopt a solution. In addition, there are also technical consequences. Two separate languages implies two spaces in which objects can reside:

Data Space for the data which is a processed by the individual tasks.

Coordination Space for data that is required to determine the control flow.

CIEL allows data to pass from the data space to the coordination space, by use of a special operator which makes assumptions on the format of the data. This feature allows for data dependent control flow. Pydron goes a step further. By avoiding a separate language, there is only one space where objects reside. Processed data and coordination data can be treated equally, as one would in a regular single-threaded program, reducing the complexity the developer has to handle to profit from parallel infrastructures.

The separation between orchestration code and computation code still exists in Pydron. Functions annotated with *@functional* contain the code executed within a task, functions annotated with *@schedule* mark orchestration code. Only orchestration code is translated into

the data-flow graph. @*functional* code might be sent to a worker node for execution, but there it is executed as regular Python code. The line between the two is less obvious in Pydron, since there is no need to use a different language for orchestration code. In addition, Pydron has the option to execute orchestration code regularly, instead of translating it into a data-flow graph (Section 6.5), further blurring the line.

Both CIEL and Pydron change the data-flow graph based on the data. CIEL allows a task to spawn new tasks during execution. To account for the dynamic nature of Python, Pydron requires additional changes, most notably it those required to remove the synchronization points (Section 5.3), We also allow tasks to trigger changes to the graph before all its inputs are available.

There are a number of features in CIEL that Pydron is currently missing, such as the multiple-queue-based scheduler, fault tolerance for the master node, and streaming. Such features will be integrated into future versions of Pydron.

Fully automated parallelization of sequential languages has been studied in depth which has lead to systems such as Helix [8]. Such systems perform a static analysis on the code to identify loops that can be parallelized. The search space for the inference of the data dependencies includes all code potentially executed within the loop. This effectively limits such approaches to the inner-most loops. Dynamic languages such as Python are particularly difficult in this respect. In consequence, parallelization is fine granular, and small orchestration overheads quickly become the bottleneck. This reflects in the way such systems operate. For example, the parallelization constructs may be directly inserted into the compiled code, instead of evaluating a data-flow graph at run-time.

Pydron parallelizes on a coarse granularity. To keep the search space reasonable, the user has to help out with the @*functional* annotation. The code analysis of Pydron is similar to the data-flow analysis performed by automatic parallelization systems, yet the design is primarliy driven by the need of coping with the dynamic nature of Python. Since Pydron can make decisions at run-time, it can avoid some of the complex problems such as pointer analysis [15]. Such analysis could still be integrated into Pydron in the future as it would allow certain decisions to be made before the actual data is computed.

We don't see Pydron as a replacement for systems such as Helix. In fact, it would be possible to combine both. Combining Pydron with another parallelization

```
@schedule
def train_forest(data, labels, count):
  forest = []
  for i in range(count):
    tree = train_tree(i, data, labels)
    forest += [tree]

  def predict(sample):
    predictions = [tree.predict(sample)[0]
                   for tree in forest]
    return Counter(predictions).most_common(1)
  return predict
```

Figure 12: Random Forest Implementation

system works very well in practice. In section 7.4 we describe how Pydron can be used with SciPy [17]. If SciPy is compiled with multi-threaded ATLAS [35], then the numerical functions would exploit multiple-cores, while Pydron can parallelize the outer loops across several machines.

9 Scalability

In this section, we demonstrate the scalability of Pydron for multi-core, cluster and cloud infrastructures. We also provide insights through several experiments on how Pydron operates. All measurements were taken with CPython 2.7.6.

9.1 Multi-core

We use a machine-learning example for the multi-core and cluster measurements. The random forest method [4] trains several decision trees on a random sub-set of the training samples. Predictions are made by majority vote among the predictions made by the individual decision trees. We used 50% of the samples in the MNIST handwritten digits data-set for training [19] (approx. 27 MB).

The code is shown in Figure 12. The *train_forest* function is annotated with @*schedule*. The *for*-loop can be unrolled completely in the beginning of the execution since *train_tree* is annotated with @*functional*. *train_forest* returns a nested function to make predictions, using a closure variable to access the forest. If predictions were expensive, then annotating the nested function with @*schedule* would parallelize the list-comprehension as well. The implementation of train_tree is using scikit-learn from SciPy [17] internally. Pydron handles calls to thrid-party libraries as any other function call (see section 7.4).

Figure 13 shows the learning time on a single machine with 64 cores (AMD Opteron 6276) when running the code using regular Python and when using Pydron

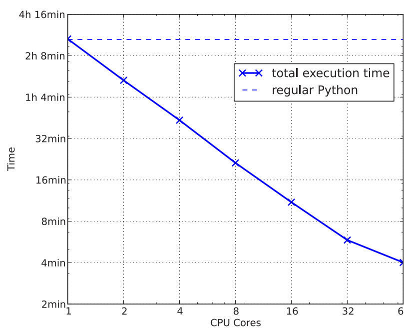


Figure 13: Random Forest Training on Multi-core

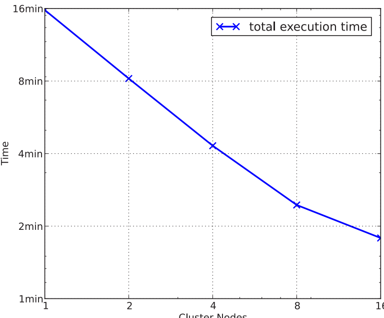


Figure 14: Random Forest Training on Cluster

with an increasing number of cores. It scales nearly linearly and only flattens once the communication overhead becomes noticeable. This is to be expected as Pydron currently makes no use of shared memory for communication. The horizontal line marks the execution time with regular, single threaded Python, which is about 90s (~1%) faster than Pydron with a single core.

9.2 Cluster

Figure 14 shows the result of the same machine-learning code when Pydron is instructed to execute it on a cluster (Intel Xeon L5520). We use the combination of the cluster and the multi-core back-end to utilize the 8 cores of each node. We run the experiment with up to 16 cluster nodes with a total of 128 cores, at which point the scalability starts to degrade due to communication overheads. The execution time of regular, single threaded, Python is not shown in the figure as it would be about eight times slower than a single node. When running Pydron with only one of the node's cores, the difference is comparable to the one shown for the 64-core machine.

```
@schedule
def parameter_sweep(in_file):
  images = []
  basis = []
  for center in np.linspace(0.01, 0.1, 6):
    for edge in np.linspace(0.7, 1.0, 6):
      images+=[create_images(in_file, center, edge)]
      basis+=[create_basis(in_file, center, edge)]
```

Figure 15: Parameter Sweep Code

9.3 Cloud

Running the machine-learning code on the cloud produces results comparable to those on the cluster, we therefore use cloud computing to demonstrate Pydron on a larger astronomy use-case.

PynPoint [2] is a method for detection of planets outside the solar system. The challenge of exo-planet detection lies in the extreme contrast between the bright host star and the faint planet. Optical effects and atmospheric distortions spread the light of the star over an area larger than the orbit of the planet. PynPoint models the point-spread function of the star with a principal component analysis (PCA) to remove the spread-out light from the star, leaving the planet visible in the residue.

We use a real high-contrast imaging data-set of β Pictoris [18] and the massive exo-planet orbiting it. The data set was taken with the Very Large Telescope. The raw data is publicly available from the European Southern Observatory (ESO) archive (Program ID: 084.C-0739(A)). Some data reduction steps [26] have already been applied to the data. The data set consists of 24000 individual exposures, totaling to 3.8 GB.

PynPoint operates in two main phases. In the first phase, the images are prepared and the basis of the PCA are calculated. In the second phase, the modeled point-spread-function of the star is removed from the exposures. The exposures are then rotated to compensate for earth's rotation and aggregated into the final result. The second phase is fast enough to be used interactively by the scientists to study the effect of the method's parameters. However, some parameters affect the first phase which takes about 15 minutes to execute. We have used Pydron to scale the parameter sweep over the two main parameters used in the first phase.

Six values are used for each parameter, resulting in a total of 36 executions. The code is shown in Figure 15. *in_file* contains the path to the input data file in HDF5 format. The two functions *create_images* and *create_basis* are both decorated with *@functional*. Since they are independent of each other, all 72 calls could be run in parallel. The implementation of those methods

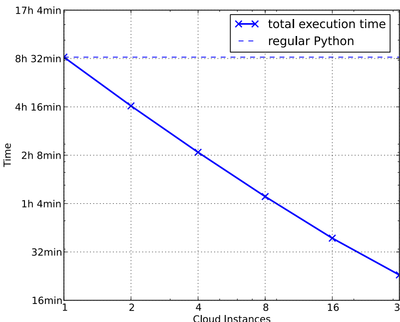


Figure 16: Scalability on Amazon EC2

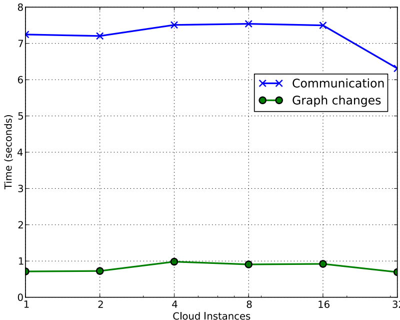


Figure 17: Pydron overhead for communication and dynamic graph changes on Amazon EC2

use numerical routines from SciPy [17] which use multi-threading internally to utilize multiple cores. Pydron can be used together with such libraries. We use Pydron's cloud back-end to parallelize across multiple cloud instances. To get a clearer performance analysis we do not combine it with the multi-core back-end. Thus we use one Python process per instance.

We use Amazon EC2 with *m2.large* instances, with two CPU cores each. Adding cores would not scale well with this workload, as the routines can only profit from the parallel SciPy library for a part of their execution. The cloud instances are connected to a shared file system used as a scratch space. This file system is provided by two separate EC2 instances (*c2.2xlarge*) which provide the storage from a total of four solid state drives. The file system is clustered with glusterFS [28]. The file system initially contains the input data.

Figure 16 shows the execution time for up to 32 instances (64 cores). The execution time includes the time required to start the instances, which takes about one minute.

Other than in the machine-learning use-case, the actual data is transmitted over a shared file system, while Pydron only handles the paths, as described in Section 4. The Pydron induced overhead is therefore very small, about 7s. With a large number of instances the throughput of the scratch file system becomes a bottleneck, as each parameter combination produces approximately 4 GB of data. This bottleneck could be easily addressed by increasing the number of nodes of the clustered file system.

The overhead introduced by Pydron is neglectable in this use-case. The translation of the Python code into the initial data-flow takes five milliseconds. Figure 17 shows that less than a second is spent for all dynamic changes in the data-flow graph and that less than eight seconds are required for communication, including serialization with pickle. Both can partially run in parallel, reducing the effective impact. With 32 instances the workers are limited by the shared file system, the lower CPU utilization speeds up pickle.

10 Conclusions

Semi-automatic parallelization provides easy-to-use access to high performance computing infrastructures for many problems that can be parallelized at a sufficiently coarse granularity.

By putting the focus on non-intrusiveness and a low learning curve, instead of on optimal usage of infrastructure, Pydron can lower the barrier for scientists to access high performance computing infrastructures.

We plan to release Pydron under an open source licence. Please check *www.pydron.org* for availability.

References

[1] apache libcloud, a unified interface to the cloud https://libcloud.apache.org, 2014.

[2] AMARA, A., AND QUANZ, S. P. Pynpoint: an image processing package for finding exoplanets. *Monthly Notices of the Royal Astronomical Society 427*, 2 (2012), 948–955.

[3] BERGÉ, J., GAMPER, L., ET AL. An ultra fast image generator (ufig) for wide-field astronomy. *Astronomy and Computing 1*, 0 (2013), 23 – 32.

[4] BREIMAN, L. Random forests. *Machine Learning 45*, 1 (2001), 5–32.

[5] BROWN, P. G. Overview of sciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 963–968.

[6] BUI, P., YU, L., ET AL. Scripting distributed scientific workflows using weaver. *Concurrency and Computation: Practice and Experience 24*, 15 (2012), 1685–1707.

[7] BUTENHOF, D. R. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[8] CAMPANONI, S., JONES, T., ET AL. Helix: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (2012), CGO '12, ACM, pp. 84–93.

[9] CATANZARO, B., KAMIL, S., ET AL. Sejits: Getting productivity and performance with selective embedded jit specialization. *Programming Models for Emerging Architectures* (2009).

[10] CHAMBERS, C., RANIWALA, A., ET AL. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 363–375.

[11] COOPER, K. D., AND TORCZON, L. *Engineering a compiler*. Morgan Kaufmann [Oxford], San Francisco (Calif.), 2012.

[12] DARTE, A., ROBERT, Y., ET AL. *Scheduling and automatic parallelization*. Springer, 2000.

[13] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI 2004* (2004), pp. 137–150.

[14] FORUM, M. P. I. MPI: A Message-Passing Interface Standard. Version 2.2, Sept. 2009.

[15] HIND, M. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (New York, NY, USA, 2001), PASTE '01, ACM, pp. 54–61.

[16] ISARD, M., AND YU, Y. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD international conference on Management of data* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 987–994.

[17] JONES, E., OLIPHANT, T., ET AL. SciPy: Open source scientific tools for Python http://www.scipy.org, 2001.

[18] LAGRANGE, A.-M., BONNEFOY, M., ET AL. A giant planet imaged in the disk of the young star beta pictoris. *Science 329*, 5987 (2010), 57–59.

[19] LECUN, Y., AND CORTES, C. The MNIST database of handwritten digits http://yann.lecun.com/exdb/mnist.

[20] MURRAY, D. G., MCSHERRY, F., ET AL. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[21] MURRAY, D. G., SCHWARZKOPF, M., ET AL. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 9–9.

[22] NVIDIA. Nvidia CUDA http://nvidia.com/cuda, 2007.

[23] OINN, T., GREENWOOD, M., ET AL. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience 18*, 10 (Aug. 2006), 1067–1100.

[24] OLSTON, C., REED, B., ET AL. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (2008), SIGMOD '08, ACM, pp. 1099–1110.

[25] PYTHON SOFTWARE FOUNDATION. pickle – python object serialization http://docs.python.org, 2014.

[26] QUANZ, S. P., KENWORTHY, M. A., ET AL. Searching for gas giant planets on solar system scales: Vlt naco/app observations of the debris disk host stars hd172555 and hd115892. *The Astrophysical Journal Letters 736*, 2 (2011), L32.

[27] RAGHAVAN, P., SHACHNAI, H., AND YANIV, M. Dynamic schemes for speculative execution of code. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998. Proceedings. Sixth International Symposium on* (Jul 1998), pp. 309–314.

[28] RED HAT INC. GlusterFS http://www.gluster.org, 2013.

[29] REFREGIER, A., AND AMARA, A. A way forward for cosmic shear: Monte-carlo control loops. *Dark Universe Journal* (in press, http://arxiv.org/abs/1303.4739).

[30] RUBINSTEYN, A., HIELSCHER, E., ET AL. Parakeet: A just-in-time parallel accelerator for python. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism* (2012).

[31] STOLTE, E., VON PRAUN, C., ET AL. Scientific data repositories: Designing for a moving target. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), SIGMOD '03, ACM, pp. 349–360.

[32] SZALAY, A. S., GRAY, J., ET AL. Indexing the sphere with the hierarchical triangular mesh. *CoRR abs/cs/0701164* (2007).

[33] THAKAR, A. R., SZALAY, A. S., ET AL. The catalog archive server database management system. *Computing in Science and Engineering 10*, 1 (2008), 30–37.

[34] VAN ROSSUM, J., AND MOORE, P. New Import Hooks http://legacy.python.org/dev/peps/pep-0302, 2000.

[35] WHALEY, R. C., AND PETITET, A. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience 35*, 2 (February 2005), 101–121.

[36] WILDE, M., HATEGAN, M., ET AL. Swift: A language for distributed parallel scripting. *Parallel Computing 37*, 9 (2011), 633–652.

[37] YU, Y., ISARD, M., , ET AL. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), vol. 8, pp. 1–14.

[38] ZAHARIA, M., CHOWDHURY, M., ET AL. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), pp. 10–10.

[39] ZAHARIA, M., DAS, T., ET AL. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 423–438.

User-Guided Device Driver Synthesis*

Leonid Ryzhyk[1,2] Adam Walker[2] John Keys[3] Alexander Legg[2] Arun Raghunath[3]
Michael Stumm[1] Mona Vij[3]

[1]University of Toronto
[2]NICTA† and UNSW, Sydney, Australia
[3]Intel Corporation

Abstract

Automatic device driver synthesis is a radical approach to creating drivers faster and with fewer defects by generating them automatically based on hardware device specifications. We present the design and implementation of a new driver synthesis toolkit, called Termite-2. Termite-2 is the first tool to combine the power of automation with the flexibility of conventional development. It is also the first practical synthesis tool based on abstraction refinement. Finally, it is the first synthesis tool to support automated debugging of input specifications. We demonstrate the practicality of Termite-2 by synthesizing drivers for a number of I/O devices representative of a typical embedded platform.

1 Introduction

Device driver synthesis has been proposed as a radical alternative to traditional driver development that offers the promise of creating drivers faster and with far fewer defects [24]. The idea is to automatically generate the driver code responsible for controlling device operations from a behavioral model of the device and a specification of the driver-OS interface.

The primary motivation for device driver synthesis is the fact that device drivers are hard and tedious to write, and they are notorious for being unreliable [8, 13]. Drivers generally take a long time to bring to production—given the speed at which new devices can be brought to market today, it is not uncommon for a device release to be delayed by driver rather than silicon issues [33].

Automatic driver synthesis was proposed in our earlier work on the Termite-1 project [24], where we formulated the key principles behind the approach and demonstrated its feasibility by synthesizing drivers for several real-world devices. The next logical step is to develop driver synthesis into a practical methodology, capable of replacing the conventional driver development process. To this end we have to address the key problems left open by Termite-1. The most important one is the quality of synthesized drivers. While functionally correct, Termite-1 drivers were bloated and poorly structured. This made it impossible for a programmer to maintain and improve the generated code and prevented synthesized drivers from being adopted by Linux and other major OSs. Furthermore, it was impossible to enforce non-functional properties such as CPU and power efficiency.

Another critical limitation of Termite-1 was the limited scalability of its synthesis algorithm, which made synthesis of drivers for real-world devices intractable. Termite-1 got around the problem by using carefully crafted simplified device specifications, which is acceptable in a proof-of-concept prototype, but not in a practical tool.

In the present project we set out to address these limitations. After several years of research we achieved significant improvement to all components of the synthesis technology: the specification language, the synthesis algorithm and the code generator.

Despite these improvements, we had come to the conclusion that the approach taken was initially *critically flawed*. The fundamental problem, in our view, was that the synthesis was viewed as a "push-button" technology that generated a specification-compliant implementation without any user involvement. As a result, the user had to rely on the synthesis tool to produce a good implementation. Unfortunately, even the most intelligent algorithm cannot fully capture the user-perceived notion of high-quality code. While in theory one might be able to enforce some of the desired properties by adding appropriate constraints to the input specification, in our experience

*This research is supported by a grant from Intel Corporation.
†NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

creating such specifications is extremely hard and seldom yields satisfactory results.

A radically different approach was needed—one that combines the power of automation with the flexibility of conventional development, and that involves the developer from the start, guiding the generation of the driver. In many ways, synthesis and conventional development are conflicting. Hence, a key challenge was to conceive of a way that allowed the two to be combined so that the developer could do their job more efficiently and with fewer errors without having the synthesis tool get in the way.

The primary contribution of this paper is a novel *user-guided* approach to driver synthesis implemented in our new tool called Termite-2 (further referred to as Termite). In Termite, the user has full control over the synthesis process, while the tool acts as an assistant that suggests, but does not enforce, implementation options and ensures correctness of the resulting code. At any point during synthesis the user can modify or extend previously synthesized code. The tool automatically analyses user-provided code and, on user's request, suggests possible ways to extend it to a complete implementation. If such an extension is not possible due to an error in the user code, the tool generates an explanation of the failure that helps the user to identify and correct the error.

In an extreme scenario, Termite can be used to synthesize the complete implementation fully automatically. At the other extreme, the user can build the complete implementation by hand, in which case Termite acts as a static verifier for the driver. In practice, we found the intermediate approach, where most of the code is auto-generated, but manual involvement is used when needed to improve the implementation, to be the most practical.

From the developer's perspective, user-guided synthesis appears as an enhancement of the conventional development process with very powerful autocomplete functionality, rather than a completely new development methodology. This vision is implemented in all aspects of the design of Termite. In particular, input specifications for driver synthesis are written as imperative programs that model the behavior of the device and the OS. The driver itself is modelled as a source code template where parts to be synthesized are omitted. This approach enables the use of familiar programming techniques in building input specifications. In contrast, previous synthesis tools, including Termite-1, require specifications to be written in formal languages based on state machines and temporal logic, which proved difficult and error-prone to use even for formal methods experts, not to mention software development practitioners.

Most previous research on automatic synthesis, including Termite-1, considered input specifications to be "correct by definition". In contrast, we recognise that input specifications produced by human developers are likely to contain defects, which can prevent the synthesis algorithm from finding a correct driver implementation. Therefore Termite incorporates powerful debugging tools that help the developer identify and fix specification defects through well-defined steps, similar to how conventional debuggers help troubleshoot implementation errors.

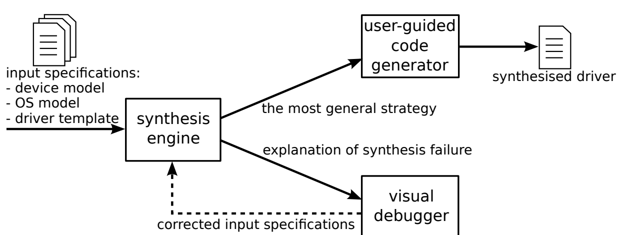


Figure 1: Termite synthesis workflow.

Another important contribution of this project is a new scalable synthesis algorithm, which mitigates the computational bottleneck in driver synthesis. Following the approach proposed in Termite-1, we treat the driver synthesis problem as a two-player game between the driver and its environment, comprised of the device and the OS. In this work, we develop this approach into the first precise mathematical formulation of the driver synthesis problem based on game theory. This enables us to apply theoretical results and algorithmic techniques from game theory to driver synthesis.

Our game-based synthesis algorithm relies on abstraction and symbolic reasoning to achieve orders of magnitude speed up compared to the current state-of-the-art synthesis techniques. The main idea of the algorithm is described in Section 4, but we refer the reader to a detailed description in an accompanying publication [30].

We evaluate Termite by synthesizing drivers for several I/O devices. Our experience demonstrates that our methodology meets our design goals, and indeed makes automatic driver synthesis practical.

Overview of Termite Figure 1 gives an overview of the driver synthesis process, described in detail in the rest of the paper. Termite takes three specifications as its inputs: a device model that simulates software-visible device behavior, an OS model that specifies the software interface between the driver and the OS, and a driver template that contains driver entry point declarations and, optionally, their partial implementation to be completed by Termite.

Given these specifications, driver synthesis proceeds in two steps. The first step is carried out fully automatically by the Termite game-based synthesis engine, which computes *the most general strategy* for the driver—a data

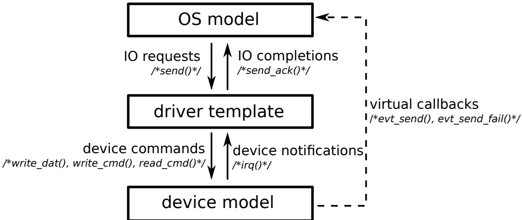


Figure 2: Input specifications for driver synthesis. Labels in italics show interfaces from the running example (Figure 3).

structure that compactly represents all possible correct driver implementations. This step encapsulates the computationally expensive part of synthesis. At the second step, the most general strategy is used by the Termite code generator to construct one specific driver implementation in C with the help of interactive input from the user.

The synthesis engine may establish that, due to a defect in one of the input specifications, there does not exist a specification-compliant driver implementation. In this case, it produces an explanation of the failure, which can be analysed with the help of the Termite debugger tool in order identify and correct the defect.

Limitations of Termite The device driver synthesis technology is still in its early days and, as such, has several important limitations. Most notably, Termite does not currently support synthesis or verification of code for managing direct memory access (DMA) queues. This code must be written manually and is treated by Termite as an external API invoked by the driver. As another example, in certain situations, explained in Section 3, Termite is unable to produce correct code without user assistance; however it is able to verify the correctness of user-provided code. We discuss limitations of Termite in more detail in Section 6.

2 Developing specifications

Input to Termite consists of the three specifications, which model the complete system consisting of the driver, the device, and the OS, shown in Figure 2. The OS and device models simulate the execution environment of the driver and specify constraints on correct driver behavior. The device model simulates software-visible device behavior. The OS model serves as a workload generator that issues I/O requests to the driver and accepts request completions in a way consistent with real OS behavior.

The virtual interface between the device and the OS, shown with the dashed arrow in Figure 2, is used by the device model to notify the OS model about important hardware events, such as completion of I/O transactions and error conditions. Methods of the virtual interface do not represent real runtime interactions between the device and the OS, but are used by the OS model to specify correctness constraints for the driver (see Section 2.3).

Finally, the driver template contains a partial driver implementation to be completed by Termite. A minimal template consists of a list of driver entrypoints without implementation. At the other extreme, it can provide a complete implementation, in which case Termite acts as a static verifier for the driver.

All specifications are written using the Termite Specification Language (TSL). In line with our goal of making synthesis as close to the conventional driver development workflow as possible, TSL is designed as a dialect of C with additional constructs for use in synthesis. We introduce relevant features of TSL throughout this section.

We minimize the amount of work needed to develop specifications for every synthesized driver by maximizing the reuse of specifications. In particular, Termite allows the use of existing device specifications developed by hardware designers in driver synthesis. Furthermore, the OS specification for the driver can be derived from a generic specification for a class of similar devices (e.g., network or storage). Thus we expect that additional per-driver effort will consist of: (1) inserting device-class callbacks in appropriate locations of the device model and (2) extending the OS specification to support device-specific features missing in the generic OS specification.

2.1 Device model

The device model simulates the device operation at a level of detail sufficient to synthesize a correct driver for it. To this end, it must accurately model external device behavior visible to software. At the same time, it is not required to precisely capture internal device operation and timing, as these aspects are opaque to the driver.

Such device models are routinely developed by hardware designers for the purposes of design exploration, simulation, and testing. They are widely used by hardware manufacturers in-house [14] and are available commercially from major silicon IP vendors [28]. These models are known as *transaction-level models* (TLMs) (in contrast to the detailed register-transfer-level models used in gate-level synthesis) [4]. A TLM focuses on software-visible events, or *transactions*, such as a write to a device register or a network packet transmission.

Existing TLMs created by hardware designers can be used with minor modifications (explained in Section 2.3) for driver synthesis. Model reuse dramatically reduces the effort involved in synthesizing a driver and is therefore crucial to practical success of driver synthesis. By reusing an existing model, we also reuse the effort invested by hardware designers into testing and debugging the model throughout the hardware design cycle, thus making driver

```
1  template dev /* Device model */
2    uint8 reg_dat, reg_cmd, reg_status = 0;
3    /* device commands */
4    controllable void write_dat(uint8 v)
5    { reg_dat = v; };
6    controllable void write_cmd(uint8 v)
7    { reg_cmd = v; };
8    controllable uint8 read_cmd()
9    { return reg_cmd; };
10   controllable uint8 read_status()
11   { return reg_status; };
12   /* internal behavior */
13   process ptx {
14     forever {
15       wait (reg_cmd == 1);
16       choice {
17         { os.evt_send(reg_dat);
18           reg_status=0; };
19         { os.evt_send_fail(reg_dat);
20           reg_status=1; };
21       };
22       reg_cmd = 0;
23       /*drv.irq(); (see Section 4*/
24     };
25   };
26 endtemplate
27
28 template os /* OS model */
29   uint8 dat;
30   bool inprogress, acked, success;
31   /* driver workload generator */
32   process psend {
33     forever {
34       dat = *; /*randomise dat*/
35       inprogress = true;
36       acked = false;
37       drv.send(dat);
38       wait(acked);
39     };
40   };
41   /* I/O completions */
42   controllable void send_ack(bool status) {
43     assert (!inprogress && !acked &&
44             status == success);
45     acked = true;
46   };
47   /* virtual callbacks */
48   void evt_send(uint8 v) {
49     assert (inprogress && v==dat);
50     inprogress = false;
51     success = true;
52   };
53   void evt_send_fail(uint8 v) {
54     assert (inprogress && v==dat);
55     inprogress = false;
56     success = false;
57   };
58   goal idle_goal = acked;
59 endtemplate
60
61 template drv /* Driver template */
62   void send(uint8 v){...;};
63   /*void irq(){...;}; (see Section 4)*/
64 endtemplate
```

Figure 3: Trivial serial controller driver specifications.

synthesis less susceptible to specification bugs. Finally, since TLMs are created early in the hardware design cycle, TLM-based driver synthesis can be carried out early as well, thus removing driver development from the critical path to product delivery.

TLMs are written in high-level hardware description languages like SystemC and DML. In order to use these models in driver synthesis, we need to convert them to TSL. This translation can be performed automatically, and we are currently working on a DML-to-TSL compiler. Since this work is not yet complete, device models used in the experimental section of this paper are either manually translated from existing TLMs or written from scratch using TLM modeling style guidelines [31].

The top part of Figure 3 shows a fragment of a model of a trivial serial controller device used as a running example. The fragment specifies the send logic of the controller, which allows software to send data characters over the serial line. The model is implemented as a TSL *template*. The template encapsulates data and code that manipulates the data, similar to a class in OOP.

The software interface of the device consists of data, command, and status registers declared in line 2. The registers can be accessed from software via the `write_dat`, `write_cmd`, `read_cmd`, and `read_status` methods (lines 4–11). The `controllable` qualifier denotes a method that is available to the driver and can be invoked from synthesized code.

The transmitter logic is modelled in lines 13–25. It is implemented as a TSL *process*. A TSL specification can contain multiple processes. The choice of the process to run is made non-deterministically by the scheduler. The process executes atomically until reaching a `wait` statement or a controllable placeholder (see below).

In line 15, the transmitter waits for a command, issued by the driver by writing value 1 to the command register. Upon receiving the command, it sends the value in the data register over the serial line. The transmission may fail, e.g., due to a serial link problem. The device signals transmission status to software by setting the status register to 0 or 1. Finally, it clears the command register, thus notifying the driver the request has completed.

Internally, the transmitter circuit consists of a shift register and a baud rate generator used to output data on the serial line. These details are not visible to software and are abstracted away in the model. We use the non-deterministic `choice` construct to choose between successful transmission and failure, without modelling the details of serial link operation. Successful and failed transmissions are modelled using `evt_send` and `evt_send_fail` events, explained in Section 2.2.

2.2 OS model

The OS model specifies the API mandated by the OS for all drivers of the given type. For example, any Ethernet driver must implement the interface for sending and receiving Ethernet packets. A separate specification is needed for each supported OS, as different OSs define different interfaces for device drivers.

Additionally, each particular device can support non-standard features, e.g., device-specific configuration options or transfer modes. These features must be added as extensions to the generic OS specification in order to synthesize support for them in the driver. TSL supports such extensions in a systematic way via the template inheritance mechanism. We do not describe this in detail due to limited space.

The OS model is written in the form of a test harness that simulates all possible sequences of driver invocations issued by the OS. The `os` template in Figure 3 shows the OS model for our running example. The main part of the model is the `psend` process. At every iteration of the loop, it non-deterministically chooses an 8-bit value (line 34) and calls the `send` method of the driver, passing this value as an argument. It then waits for the driver to acknowledge the transmission of the byte (line 38) before issuing another request. The driver acknowledges the transmission via the `send_ack` callback (line 42). The callback sets the `acked` flag, which unblocks the `psend` process.

We keep the specification concise by modeling the state of the driver-OS interface, as opposed to the internal OS state and behavior. For example, the `acked` variable (line 30) serves to model the flow of data between the OS and the driver and is not necessarily present in the OS implementation.

2.3 Connecting device and OS models

In addition to simulating I/O requests to the driver, the OS model also specifies the semantics of each request in terms of device-internal events that must occur in order to complete the requested I/O operation. In our running example, after the OS invokes the `send` method of the driver and before the driver acknowledges completion of the request, the device must attempt to send the requested data over the serial line. This requirement establishes a connection between the device and OS models and must be specified explicitly in order to enable Termite to generate a driver implementation that correctly handles the OS request. Note that we only need to specify *which* hardware events must occur, but not *how* the driver generates them.

In order to develop such specifications, we need a way to refer to relevant state and behavior of the device from the OS model. At the same time, in order to maximize specification reuse, we would like to keep the OS specification device-independent. To reconcile these conflicting requirements, we introduce a *virtual interface* between the device and OS model. This interface consists of callbacks used by the device model to notify the OS model about important hardware events. The virtual interface does not represent real runtime interactions between the device and the OS, but serves as part of the correctness specification.

We define a virtual interface for each class of devices. Such *device-class* interfaces are both device and OS-independent. The device-class interface can be extended with additional device-specific callbacks as required to specify a driver for a particular device.

In our example, we define a device-class interface consisting of two virtual callbacks: `evt_send` and `ev_send_failed`, invoked respectively when the device successfully transmits and fails to transmit a byte. These callbacks are invoked in lines 17 and 19 of the device model. The `evt_send` handler is shown in line 48 of the OS model. The assertion in line 49 specifies that the send event is only allowed to occur if there is an outstanding send request in progress and the value being sent is the same as the one requested by the OS. We reset the `inprogress` flag to false in line 50, thus marking the current request as completed; line 51 sets the `success` flag to true, thus indicating that the transfer completed without an error. The `evt_send_fail` handler is identical, except that it sets the `success` flag to false. The flags are checked by the `send_ack` method, which asserts that the driver is only allowed to acknowledge a completed request (`!inprogress`) that has not been acknowledged yet (`!acked`) and that the completion status reported by the driver must match the one recorded in the `success` flag.

In this example we use C-style assertions to rule out invalid system behaviors. Assertions alone do not fully capture requirements for a correct driver behavior. For example, a driver that remains idle does not violate any assertions. Hence, we need to specify requirements for the driver to make forward progress. We introduce such requirements into the model in the form of *goal conditions*, that must hold *infinitely often* in any run of the system. For example, a goal may require that the driver is infinitely often in an idle state with no outstanding requests from the OS. The OS can force the driver out of the goal by issuing a new I/O request. To satisfy the goal condition, the driver must return to the goal state by completing the request. Line 58 in Figure 3 defines such a goal condition that holds whenever the `acked` flag is set, i.e., the driver has no unacknowledged send requests.

2.4 Driver template

The bottom part of Figure 3 shows the driver template for the running example consisting of a single `send` entry point invoked by the OS. The ellipsis in line 62 represent a location for inserting synthesized code and are part of TSL syntax. We refer to such locations as *controllable placeholders*.

3 User-guided code generation

The set of input TSL specifications is fed into the Termite synthesis engine, which then automatically computes the most general strategy for the driver. Given a state of the system, the most general strategy determines the set of all valid driver actions in this state. The most general strategy is used by the Termite code generator to produce a driver implementation in C in a user-guide fashion.

The Termite code generator GUI is similar to a traditional integrated development environment with two additional built-in tools: the *generator* and the *verifier*. The generator works as advanced auto-complete that helps the user to fill the controllable placeholders inside the driver template with code. At any point, the user can invoke the generator to synthesize a single statement or a complete block of code inside a controllable placeholder via a mouse click on the target code location. The user can arbitrarily modify and amend the generated code. However, the generator never modifies user code. Instead it tries to extend it to a complete implementation, which is always possible provided that the existing code is consistent with the most general strategy. The generator currently only allows synthesizing statements after the last control location within a branch. However this restriction is not a conceptual one and will be lifted by ongoing development.

The verifier automatically and on the fly checks that the driver implementation, comprised of a mix of generated and manually written code, is consistent with the most general strategy, thus maintaining strong correctness guarantees that one would expect in automatically synthesized code. The verifier symbolically simulates execution of the system, following the partial driver implementation created so far, and signals the user whenever it encounters a transition that violates the most general strategy.

In the first approximation, the generator algorithm is quite simple: given a source code location, it determines the set of possible system states in this location, picks an action for each state from the most general strategy and translates this action into a code statement. In practice the algorithm uses a number of heuristics to produce compact and human-readable code. In particular, whenever there exists a common action in all possible states in the given location, the algorithm produces straight-line code without branching. For example, when running the generator on the specification in Figure 3, it automatically generates the following code for the `send` function (line 62):

```
void send(uint8 v){
    dev.write_dat(v);
    dev.write_cmd(1);
    wait(dev.reg_cmd==0);
    if (os.success) {
        os.send_ack(true);
    } else {
        os.send_ack(false);
    };}
```

This implementation correctly starts the data transfer by writing the value to be sent to the data register and setting the command register to 1. It then waits for the transfer to complete, which is signalled by the device by resetting the command register to 0. Finally, it acknowledges the completion of the transfer to the OS.

Note that the generated code refers to the `dev.reg_cmd` and `os.success` variables. These variables model internal device and OS state respectively and cannot be directly accessed by the driver. This example illustrates an important limitation of Termite—it assumes a white-box model of the system, where every state variable is visible to the driver. Ideally, we would like to synthesize an implementation that automatically infers the values of important unobservable variables. In this case, the value of the command register can be obtained by the driver by executing the `read_cmd` action. Furthermore, the value of the `os.success` variable is correlated with the completion status of the last transfer, which can be obtained by reading the device status register.

While Termite currently cannot produce such an implementation automatically, it implements a pragmatic tradeoff that helps the user build and validate a correct implementation with modest manual effort. The code generator warns the user that the auto-generated code accesses private variables of the device and OS templates. This prompts the user to provide a functionally equivalent valid implementation, replacing the `wait` statement with a polling loop and using the `read_status` method to check transfer status:

```
void send(uint8 v){
    dev.write_dat(v);
    dev.write_cmd(1);
    while(dev.read_cmd()==1);
    if (dev.read_status()) {
        os.send_ack(true);
    } else {
        os.send_ack(false);
    };}
```

The verifier automatically checks the resulting implementation and confirms that it satisfies the input specification.

Note that in this example we have synthesized code

that correctly handles device errors. This was possible, as our input device specification correctly captures device failure modes (namely, transmission failure) and our OS specification describes how the driver must report errors to the OS (via the `status` argument of the completion callback).

In principle, it is also possible to synthesize a driver implementation that handles device and OS failures *not* captured in the specifications: since the synthesis tool knows all possible valid environment behaviors, it can easily detect invalid behaviors and handle them gracefully. Automatic synthesis of such *hardened* device drivers is a promising direction of future research.

The final step of the code generation process translates the synthesized driver implementation to C. This is a trivial line-by-line translation. We expect this translation to become unnecessary in the future as our ongoing work on the TSL syntax aims to make the synthesized subset of TSL a strict subset of C.

Maintaining synthesized code Device driver development is not a one-off task: following the initial implementation, drivers are routinely modified to implement additional functionality, adapt to the changing OS interface or support new device features.

The user-guided code generation method naturally supports such incremental maintenance. A typical maintenance task proceeds in three steps. First, the developer amends device and OS models to reflect the new or changed functionality. Second, they add new methods to the previously synthesized driver, if necessary, and replace existing driver code that is expected to change with a controllable placeholder. Finally, the user runs Termite to synthesize code for all controllable placeholders. Termite treats all existing driver code as part of the uncontrollable environment. Hence, if some of the old code is incorrect in the context of the new specifications, this will lead to a synthesis failure, and counterexample-based debugging is used to identify the faulty code, as described in Section 5.

As an example, we synthesize a new version of the driver for our running example assuming a more advanced version of the serial controller device that uses interrupts to notify the driver on completion of a data transfer. The new device model is obtained by uncommenting line 23 of the device model in Figure 3, which invokes the interrupt handler method of the driver after each transfer. The driver template is extended with the `irq` method (line 63). We use the previously synthesized implementation of the `send` method, but manually remove the last two lines, which implement polling, as we want the new implementation to use interrupts instead:

```
void send(uint8 v){
    dev.write_dat(v);
    dev.write_cmd(1);}
```

Finally, we run Termite on the resulting specifications and use the generator to automatically produce the following implementation of the new `irq` method:

```
void irq(){
    if (os.success) {
        os.send_ack(true);
    } else {
        os.send_ack(false);
    };}
```

As before, we manually replace the if-condition in the first line with

```
if (dev.read_status())
```

This example illustrates how Termite supports incremental changes to the driver by reusing previously synthesized code, while maintaining strong correctness guarantees.

Instrumenting synthesized code Termite does not automatically instrument synthesized code for debugging, logging, accounting, etc. However, the user can add such instrumentation manually. Termite interprets such code as no-ops and, as with any manual code, never makes any modifications to it.

4 Synthesis

In this section we give a high-level overview of the Termite synthesis algorithm. We refer the reader to the accompanying publication [30] for a detailed description.

4.1 Driver synthesis as a game

We formalize the driver synthesis problem as a *two-player game* [29] between the driver and its environment. The game is played over a finite automaton that represents all possible states and behaviors of the system. Transitions of the automaton are classified into *controllable* transitions triggered by the driver and *uncontrollable* transitions triggered by the device or OS. A winning strategy for the driver in the game corresponds to a correct driver implementation. If, on the other hand, a winning strategy does not exist, this means that there exists no specification-conforming driver implementation.

Two-player games naturally capture the essence of the driver synthesis problem: the driver must enforce a certain subset of system behaviors while having only partial control over the system.

Figure 4 illustrates the concept using a trivial game automaton that models the core of our running example. Controllable and uncontrollable transitions of the automaton are shown with solid and dashed arrows respectively.

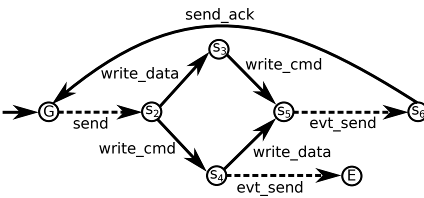


Figure 4: A simple two-player game.

The goal of the driver in the game is to infinitely often visit the initial state, labelled `G`, which represents the situation when the driver does not have any outstanding requests. After getting a `send` request from the OS, the driver must write data and command registers to start the data transfer. Writing the command register first may trigger a hardware send event before the driver has a chance to write the data register. As a result, wrong data value gets sent, taking the game into an error state `E`. Hence, state s_4 is losing for the driver. To avoid this state, the correct strategy for the driver is to play `write_data` in state s_2, followed by `write_cmd`. In s_5 the driver must remain idle until the environment executes the `evt_send` transition.

Games and strategies Formally, a two-player game $G = \langle S, I, L_c, L_u, \delta_c, \delta_u, \Phi \rangle$ consists of a set of states S, a subset of initial states $I \subseteq S$, sets of controllable and uncontrollable actions L_c and L_u, controllable transition relation $\delta_c \subseteq S \times L_c \times S$, uncontrollable transition relation $\delta_u \subseteq S \times L_u \times S$, and a game objective $\Phi \subseteq S^\omega$ (where S^ω represents the set of infinite sequences of states in S).

The game proceeds in rounds, starting from an initial state. In each round, in state s, both players select actions l_c and l_u available to them in s, and the game transitions non-deterministically to one of the states in $\delta_c(s, l_c) \cup \delta_u(s, l_u)$. Intuitively, the system scheduler chooses the player to make a move at each round. The scheduler can be thought of as part of the uncontrollable environment. Note that this is different from turn-based games like chess, where players strictly alternate in making moves. In the example in Figure 4, the driver can avoid the error state by choosing the `write_dat` action in state s_4; however the environment can override this choice by playing `evt_send`.

The infinite sequence of states $(s_0, s_1, ...) \in S^\omega$ visited by the game is called a *run*. A *strategy* for the driver player is a function $\pi : S \rightarrow 2^{L_c}$ that maps each state of the game into a set of actions to play in this state. The strategy determines a set $Outcomes(I, \pi) \subseteq S^\omega$ of all possible runs generated by the driver choosing one of the actions in $\pi(s)$ in every state s in the run.

Given a state s and a strategy π such that $Outcomes(\{s\}, \pi) \subseteq \Phi$, we say that s is a *winning state* for the driver, π is a winning strategy in s, and actions in $\pi(s)$ are *winning moves* in s. The game G is winning for the driver if all states in I are winning. The *most general winning strategy* maps every winning state s to a set of all winning moves in s, and all other states to an empty set.

In Termite we use game objectives of a particular form, called *generalised reactivity-1* (GR-1) objectives [22]. Such an objective consists of a finite set $\{B_1, ...B_n\}$, $B_i \subseteq S$ of *goal sets* and a finite set $\{F_1, ...F_k\}$, $F_i \subseteq S$ of *fair sets*. A winning strategy for the driver must make sure that the game infinitely often visits each of the goal sets, provided that the environment guarantees that the game does not get stuck in a fair set forever.

Intuitively, a goal set represents a constraint on the driver behavior, requiring the driver to force the game into the goal infinitely often, while a fair set represents a constraint on the environment, preventing it from staying in certain states forever. The game in Figure 4 has a single goal set $B_1 = \{g\}$ and a single fair set $F_1 = \{s_4, s_5\}$, i.e., the driver must acknowledge each send request from the OS, provided that the environment eventually performs the `evt_send` action after it has been enabled.

4.2 TSL compiler

In order to compute the most general driver strategy as a solution of a two-player game, we must first convert input TSL specifications into a game automaton. This conversion is performed by the TSL compiler.

Real driver specifications have large state spaces, which cannot be feasibly represented by explicitly enumerating states, as in Figure 2. Therefore, in Termite we represent games symbolically. The state space of the game is defined in terms of a finite set of state variables X, with each state $s \in S$ representing a valuation of variables in X. The TSL compiler introduces a state variable for each TSL variable declared in one of the input templates. In addition, auxiliary state variables are introduced to model the current control location of each TSL process.

We model controllable and uncontrollable actions as valuations of action variables Y_c and Y_u. Transition relations δ_c and δ_u are represented symbolically as formulas over state variables X, action variables Y_c and Y_u, and next-state variables X'.

The TSL compiler splits the input specification into controllable and uncontrollable parts and translates them into controllable and uncontrollable transition relations respectively. The controllable part is comprised of controllable methods that can be invoked by the driver. The controllable transition relation δ_c is computed by rewriting controllable methods in the *variable update form*. Consider, for example, variable `reg_dat` declared in line 2 in Figure 3. This variable is only modified by the `write_dat`

method in line 4. The corresponding fragment of the controllable transition relation in the variable update form is $\texttt{reg_dat}' := (\texttt{tag} = \texttt{write_dat})\ ?\ \texttt{v} : \texttt{reg_dat}$, where $\texttt{reg_dat}'$ is the next-state variable representing the value of reg_dat after the transition, and tag and v are controllable action variables, where tag models the method being invoked, and v is the argument of the method.

The uncontrollable part of the specification is comprised of TSL processes, which model device and OS behavior. We syntactically decompose each process into atomic transitions. Recall that a process executes atomically until reaching a wait statement or a controllable placeholder. Consider the ptx process in line 13 in Figure 3. The process is initially paused in the wait statement. It is scheduled to run when the wait condition holds. It executes the statements in lines 16–22 atomically and stops again in line 15. As part of this atomic transition, the process sets the reg_cmd variable to 0 (line 22). This is the only uncontrollable transition that modifies this variable, hence the uncontrollable update function for this variable is defined as follows: $\texttt{reg_cmd}' := (\texttt{reg_cmd} = 1 \wedge \texttt{pid} = \texttt{ptx})\ ?\ 0 : \texttt{reg_cmd}$, where pid is an uncontrollable action variable that models the scheduler's choice of a process to run, and the $\texttt{reg_cmd} = 1$ conjunct corresponds to the wait condition in line 15.

Finally, we need to generate the game objective Φ. In a symbolic representation of the game, goal and fair sets are specified as conditions over state variables that hold for each state in the set. The TSL compiler outputs a goal set B_i for each goal declared in the input specification and a fair set F_i for each wait statement. The latter guarantees that every runnable process gets scheduled eventually.

In addition to goal conditions, a TSL specification also contains assertions, which must never be violated. We model assertions using an auxiliary boolean state variable ε, which is set to true whenever an assertion is violated and remains true forever after. We add an extra constraint $\varepsilon = false$ to each accepting set B_i. An assertion violation permanently takes the game out of B_i, and therefore can not occur in any winning run of the game.

4.3 Solving the game

The Termite game solver takes a game automaton produced by the TSL compiler, determines whether all initial states of the system are winning and, if so, computes the most general winning strategy for the game. A successful approach to solving two-player games with GR-1 objectives was proposed by Piterman et al. [22]. We give an overview of their algorithm and briefly explain how we extend it to address the scalability bottleneck.

Algorithm 1 Computing the set of winning states

> **function** REACH(B)
> $Y \leftarrow \varnothing$
>   **loop**
> $Y' \leftarrow CPre(Y \cup B)$
>   **if** $Y' = Y$ **return** Y
> $Y \leftarrow Y'$
>
> **function** WINNINGSET($\{B_1, ..., B_n\}$)
> $Z \leftarrow S$
>   **loop**
> $Z' \leftarrow \bigcap_{i=1..n} \text{REACH}(Z \cap B_i)$
>   **if** $Z' = Z$ **return** Z
> $Z \leftarrow Z'$

The algorithm is based on exhaustive exploration of the state space of the game. Given a goal set B, we first determine the set of states from which the driver can force the game into B in one step, called the *controllable predecessor* of B. The controllable predecessor consists of all states s that satisfy both of the following conditions:

1. All uncontrollable transitions available in s lead to some state in B. Hence, if the scheduler chooses to execute an uncontrollable transition, it is guaranteed to take the game to B.
2. There exists at least one winning controllable transition from s to B **or** s belongs to a fair region. In the former case, the driver must perform the winning transition; in the latter case it must remain idle waiting for an uncontrollable transition, which is guaranteed to occur due to fairness.

Having computed the controllable predecessor of B, we apply the controllable predecessor operator again to the resulting set, thus obtaining the set of states from which the driver can force the game into the goal within two steps. We repeat until no new states can be discovered, at which point we have found all states from which the driver can force the game into the goal in a finite number of rounds. This computation is performed by the REACH function shown in Algorithm 1.

Recall that a GR-1 game can have multiple goal regions, and in order to win the game the driver must visit each goal region B_i infinitely often. Using the REACH function, we compute the set $Z = \bigcap_i \text{REACH}(B_i)$, from which any of the goals can be reached at least once. Next, we compute $Z' = \bigcap_i \text{REACH}(Z \cap B_i)$. It is easy to see that Z' contains all states from which any of the goals can be reached twice. Furthermore, by construction, $Z' \subseteq Z$. By continuing the last computation until a fixed point is reached, we obtain all winning states of the game, as shown in function WINNINGSET (Algorithm 1).

The algorithm presented above is polynomial in the size of the game automaton. We have developed a highly opti-

mized implementation of the algorithm, which uses symbolic data structures [3] to compactly represent large sets of states and transitions. Nevertheless, when applying it to games arising in driver synthesis, we hit a computational bottleneck due to a state explosion.

We overcome this bottleneck by using abstraction to reduce the dimensionality of the problem. The particular form of abstraction used by Termite is *predicate abstraction* [12], where concrete state variables of the game are replaced with boolean predicates over the original variables. Abstraction is adaptively refined by introducing new predicates that capture important relations among concrete variables. The predicate-based abstraction-refinement algorithm for games is one of the key technical contributions of Termite. It is described in detail in an accompanying paper [30].

4.4 Verification as a special case of synthesis

Consider the situation where not only the OS and the device, but also the driver behavior is fully specified, so that the synthesizer does not have any freedom to pick driver actions. If the resulting game is winning for the driver, i.e., every possible run of the game satisfies the objective, then the provided driver implementation is correct. Thus, verification can be seen as a special case of the synthesis problem where all transitions in the system are uncontrollable. Hence, our game solving algorithm doubles as a driver verification algorithm. Termite also supports hybrid scenarios: given a partially implemented driver with placeholders for synthesized code, it determines whether the given partial implementation can be extended to a complete one and, if so, fills out the placeholders in the user-guided fashion.

5 Debugging with counterexamples

An important practical issue in game-based synthesis is the complexity of diagnosing synthesis failures due to defects in the input specifications. In the event that Termite fails to solve the game, the user needs to trace the failure back to the specification defect. However, the failure does not carry any information about the defect, which makes the problem harder to resolve.

In Termite we propose a new approach to troubleshooting synthesis failures based on the use of *counterexample strategies*. A counterexample strategy is a strategy on behalf of the environment that prevents the driver from winning the game. It is obtained by solving the *dual game*, where, in order to win, the environment must permanently force the game out of one of the goal regions. A winning strategy in the dual game is guaranteed to exist whenever solving of the primary game fails.

In order to detect and fix the defect in an input specification, the driver developer relies on their understanding of the OS and device logic. The role of the counterexample strategy is to guide the developer towards the defect. To automate this process, we developed a powerful visual debugging tool that allows the user to interactively simulate intended driver behavior and observe environment responses to it. The user plays the game on behalf of the driver, while the tool responds on behalf of the environment, according to the counterexample strategy.

In a typical debugging session, the debugger, following the counterexample strategy, generates a sequence of requests that are guaranteed to win against the driver. The user plays against these requests by specifying device commands that, they believe, represent a correct way to handle the request. Since this sequence of requests *cannot* be handled correctly given the current input specification, at some point in the game the user runs into an unexpected behavior of one of the players, e.g., one of user-provided commands does not change the state of the device as expected or the environment performs an uncontrollable transition that violates an assertion. Based on this information, the user can revise the faulty specification.

At every step of the interactive debugging session, the debugger either chooses a spoiling uncontrollable action based on the counterexample strategy or, if the system is inside a controllable placeholder, allows the user to choose a controllable action to execute on behalf of the driver. In the former case the spoiling uncontrollable action corresponds to a transition in one of the TSL processes. The user can explore this transition by stepping through it, exactly as they would in a conventional debugger. In the latter case, the user provides the action that they would like to perform by typing and executing corresponding code statements.

The tool supports a number of features aimed to make the debugging process as simple as possible for the user. We mention two of them here. First, the debugger interactively prompts actions available to the driver at each step. Second, the debugger keeps the entire history of the game and allows the user to go back to one of previously explored states and try a different behavior from there.

6 Limitations of Termite

In Section 3, we described one limitation of Termite, namely the lack of support for grey-box synthesis. In this section we discuss other limitations, which, we hope, will help define the agenda for continuing research in driver synthesis.

Most importantly, Termite does not currently support automatic synthesis of direct memory access (DMA)

management code. Many modern devices transfer data directly to and from main memory, where it is buffered in data structures such as circular buffers and linked lists. These data structures can have very large or infinite state spaces and cannot be easily modeled within the finite state machine-based framework of Termite. Efficient synthesis for DMA requires enhancing the synthesis algorithm to use more compact representation of DMA data structures, which is the focus of our ongoing research. At this time, code for manipulating DMA data structures must be written manually. This code is not interpreted or verified by Termite. For example, we use this approach to synthesize a DMA-capable IDE disk driver (Section 7).

Device drivers in modern OSs contain a significant amount of boilerplate code that is not directly related to the task of controlling the device. This includes binding the driver to I/O resources (memory mapped regions, interrupts, timers), registering the driver with various OS subsystems, allocating DMA memory regions, creating sysfs entries, etc. While much of this functionality could be synthesized within the game-based framework, we do not believe that this is the correct approach. Previous research has demonstrated that this boilerplate code can be generated in a principled way from declarative specifications of the driver's requirements and capabilities [26]. This technique has lower computational complexity than game solving and better captures the essence of the task. A practical driver synthesis tool can combine game-based synthesis of the core driver logic responsible for controlling the device with declarative synthesis of boilerplate code. As a result, the current version of Termite assumes this boilerplate code is written manually as a wrapper around the synthesized driver.

Drivers execute in a concurrent OS environment and must handle invocations from multiple threads, as well as asynchronous hardware interrupts. We separate synthesis for concurrency into a separate step. Drivers synthesized by Termite are correct assuming a sequential environment, where driver entry points are invoked atomically. The resulting sequential driver is then processed by a separate tool that performs a sequence of transformations of the driver source code, which preserve the driver's sequential behavior, while making the driver thread-safe. Such transformations include adding locks around critical code sections, inserting memory barriers, and reordering instructions to avoid race conditions. Concurrency synthesis is still work in progress and is beyond the scope of this paper. Our preliminary results are published in [5, 6].

Termite does not explicitly support specification and synthesis of timed behaviors. Instead, it uses a pragmatic approach that allows it to synthesize time-sensitive behavior without having to explicitly reason about time. To this end, Termite conservatively approximates timed operations by fairness constraints: it ignores the exact duration of each device operation, but keeps the knowledge that the operation will complete *eventually*, and synthesizes a driver that waits for the completion. Termite is also able to handle time-out conditions, modeled as external events. However, at this time it is not capable of generating device drivers for hard real-time systems, where the driver must guarantee completion of I/O operations by a certain deadline.

7 Implementation and evaluation

The version of Termite presented here consists of 30,000 lines of Haskell code. The estimated overall project effort is 10 person years. Termite is available in source and binary form from the project webpage[1].

We evaluate Termite by synthesizing drivers for eight I/O devices. Specifically, we synthesized drivers for a UVC-compliant USB webcam, the 16550 UART serial controller, the DS12887 real-time clock, and the IDE disk controller for Linux, as well as seL4 [16] drivers for I2C, SPI, and UART controllers on the Samsung exynos 5 chipset[2] and SPI controller on the STM32F10 chipset. With the exception of the IDE disk, these devices are representative of peripherals found in a typical embedded platform, such as a smartphone. Our synthesized drivers implement data transfer, configuration and error handling. The main barrier to synthesizing drivers for more advanced devices, e.g., high-performance network controllers, is the current lack of support for synthesis of DMA code in the current version of Termite.

Modelling complexity Models of UART and DS12887 devices were developed based on existing publicly available device models [32, 20]. Models of other devices were derived from their vendor-provided documentation, following standard TLM modeling guidelines [31]. OS models for the relevant device classes were created based on Linux kernel documentation and source code.

Table 1 summarises the size, in lines of code, of device and OS models in our case studies. Developing a complete set of specifications for each driver took approximately one week, of which only one to three days were spent building the models and the rest of the time was spent studying device and OS documentation. This efficiency can be attributed to the choice of the right level of

[1] `http://termite2.org`

[2] At the time of writing, the exynos drivers have not yet been tested due to hardware availability issues; however we confirmed via manual inspection that they implement the same device control sequences as existing manually developed drivers.

	input spec		driver	
	OS	device	synthesized	native
webcam	102	385	113	307
16450 UART	122	167	74	261
exynos UART	128	252	37	166
STM SPI	73	244	24	64
exynos SPI	88	239	40	183
exynos I2C	146	180	79	211
RT clock	118	252	84	183
IDE	188	480	94[a]	474

[a]Excluding 36 lines of manually written code that manipulates the DMA descriptor table.

Table 1: Size (in lines of code) of input specifications and of synthesized and equivalent manually written drivers.

	vars(bits)	refine-ments	predi-cates	synt. time (s)	verif. time (s)
webcam	128 (125565)	47	192	215	794
16450 UART	81 (407)	65	128	210	464
exynos UART	80 (1185)	54	111	645	82
STM SPI	68 (389)	29	63	67	31
exynos SPI	83 (933)	31	72	25	44
exynos I2C	65 (303)	21	56	45	96
RT clock	92 (810)	25	74	56	127
IDE	114 (1333)	42	105	285	778

Table 2: Performance of the Termite game solver.

abstraction and modeling language. In particular, the use of transaction-level device modeling abstracts away complicated internal device machinery by focusing on high-level events relevant to driver synthesis, while the TSL language allows modeling the driver environment using standard programming techniques, as illustrated by our running example.

Interestingly, we found the most error-prone step in developing specifications for driver synthesis to be defining correct relative ordering of OS-level and device-level events with the help of the virtual interface (Section 2.3). Naïve specifications tend to be either too restrictive, leading to synthesis failures, or too liberal, leading to incorrect synthesized drivers. As we gained more experience synthesizing different types of drivers, we identified common modeling patterns that help avoid errors in virtual interface specifications.

As a common example, most virtual interfaces contain callbacks that signal a change to one of device configuration parameters, e.g., transfer speed, parity, etc. A naïve OS model may only allow such a callback to be triggered when the OS has requested a change to the corresponding device setting. However, many devices only allow setting multiple configuration parameters simultaneously, so that setting any individual parameter triggers multiple callbacks, thus making the specification non-synthesizable. The problem can be rectified by changing the device specification to only trigger callbacks if the new value of the parameter is different from the old one; however this bloats the device model due to the extra checks. A better solution, used in all our models, is to design the OS specification to allow configuration callbacks to be triggered at any time, provided that the new value of the parameter is equal to the last value requested by the OS.

Synthesis time Table 2 summarises the performance of the Termite game solver in our case studies. The second column of the table characterises the complexity of the two-player game constructed by the TSL compiler from the input specifications in terms of the number of states variables and the total number of bits in these variables. The third column shows the number of iterations of the abstraction refinement loop required to solve the game. The next column shows the size of the abstract game at the final iteration, in terms of the number of predicates in the abstract state space of the game. These results demonstrate the dramatic reduction of the problem dimension achieved by our abstraction refinement method. The second-last column shows that the Termite game solver was able to find the most general winning strategy within a few minutes in all case studies.

We compared the performance of the Termite game solver against a state-of-the-art abstraction refinement algorithm for games [10] as well as against the standard symbolic algorithm for solving games without abstraction [22]. In all case studies, the Termite solver was the only one to find a winning strategy within a two-hour limit. We refer the reader to [30] for a more detailed performance analysis of the Termite synthesis algorithm.

The final column of Table 2 shows the time that it took Termite to verify a complete driver. Recall that the Termite synthesis algorithm doubles as a verification algorithm and can be used to verify drivers written in TSL. We used complete synthesized drivers, containing a combination of manual and automatically generated code, as inputs to Termite. We have been able to successfully verify all of our drivers. We also experimented with introducing faults to synthesized drivers. Termite was able to detect these faults and produce correct counterexample strategies. In most cases verification took longer than synthesis. The reason for this is that Termite has not yet been optimized for verification workloads. This is one area for future improvement.

User-guided code generation and debugging We evaluate the key contribution of this paper, namely the user-guided debugging and code generation technique. Each line of code in a Termite-generated driver originates from one of three sources: it can be (1) synthesized automatically by the tool, (2) developed offline and given to Termite as part of the driver template, or (3) added or modified by the user during an interactive code generation ses-

sion. A perfect synthesis tool, capable of generating a complete driver fully automatically while producing code that meets all non-functional requirements, would eliminate the need for manual code altogether. We do not believe that such a tool is feasible in the near future. We therefore explore the tradeoffs that arise when using our current, imperfect, tool. In particular, we would like to empirically characterize situations when the user can rely on the synthesizer to automatically produce near-optimal code, and when they are better off completely or partially implementing certain functionality manually. These tradeoffs are likely to change as the tool improves.

Based on our experience so far, automatic synthesis is most helpful in generating code that performs device configuration or starts a data transfer. This code may involve a long sequence of commands to the device, which must be issued in the right order and with correct arguments. The synthesis algorithm of Termite proved more effective at doing this than human developers, producing correct code that only requires minimal cosmetic changes in most cases. For example, Figure 5 shows a screenshot of Termite with a synthesized implementation of the IDE driver `write()` function, which starts a data transfer to the device. The function writes request parameters into appropriate device data registers and sets bit fields in command registers to prepare the device for data transfer. One deficiency in this auto-generated implementation is that it uses absolute values instead of symbolic constants for bit fields.

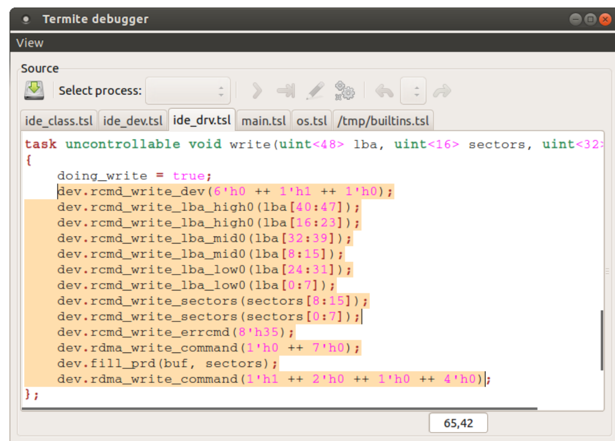


Figure 5: Screenshot of Termite with a synthesized implementation of the IDE driver. Automatically generated code is highlighted.

As another example of suboptimal synthesized code, consider the following synthesized fragment

```
void packet_received() {
  if (((packet_data[9:9] == 1) &&
       (packet_data[14:14] == 1))) {
    os.ack_packet(1,1,packet_data[16:32]);
  } else if ((dev.packet_data[9:9] == 1)) {
    os.ack_packet(1,0,packet_data);
  } else if ((dev.packet_data[14:14] == 1)) {
    os.ack_packet(0,1,packet_data[16:32]);
  } else {
    os.ack_packet(0,0,packet_data[16:32]);
  };
};
```

which can be replaced by an equivalent one-liner

```
os.ack_packet(packet_data[9:9],
    packet_data[14:14],packet_data[16:32]);
```

While both issues can, and will, be addressed by an improved code generation algorithm, our experience shows that unaccounted corner cases will arise occasionally. Therefore, the ability to manually modify synthesized code without sacrificing correctness is crucial for a practical synthesis tool.

Limitations of Termite are most noticeable in synthesizing interrupt handler code responsible for processing I/O completions. This involves querying device state to determine which operations completed and with what status, reporting results to the OS, and clearing interrupt status registers. Since Termite does not support grey-box synthesis, it can not generate this code automatically and instead produces code that directly accesses device-internal state (see Section 3). Termite correctly reports such situations and allows the user to mitigate them by manually editing synthesized code. In practice, however, we found it easier to develop most of the interrupt handler logic offline, as part of the driver template, and rely on Termite to (a) establish correctness of this code and (b) extend it to a complete implementation.

In our case studies, 60% to 90% of the code was generated fully automatically, with the rest of the code produced in a user-guided fashion. Once an initial version of device and OS specifications was ready, it took us several hours to generate the driver implementation for each of our case studies. Three quarters of this time was spent debugging the input specifications, with the rest of it spent generating driver source code with the help of the user-guided code generation GUI.

We found counterexample-driven debugging to be crucial to the productivity of synthesis-based development. Before the debugger was available, we had to rely on code inspection to identify defects in the input specifications, which proved to be a frustrating and unpredictably long process. The Termite debugger streamlines this process, giving us the confidence that any failure can be localised by following well-defined steps. A typical debugging session takes a few minutes and involves entering only a few commands manually before the defect is localised.

Size of synthesized code The last two columns of Table 1 compare the size of synthesized drivers to existing manually developed drivers. Synthesised drivers are significantly more compact than conventional drivers for two main reasons. First, as explained in Section 6, we only synthesize the driver logic directly responsible for controlling the device. Conventional drivers typically contain a large amount of boilerplate code managing various OS resources. We believe that this code can and should be synthesized using complementary techniques. At the moment we implement this functionality manually as a wrapper around the synthesized driver.

Second, conventional device drivers are often designed to support multiple similar devices with slightly different interfaces and capabilities. This leads to code bloat, as the driver must implement multiple versions of various operations, as well as logic to dynamically discover device capabilities and choose the right implementation to use. In contrast, every Termite driver supports one specific device model with a fixed set of features. Drivers for similar devices can share common specification code, but are synthesized as separate source code modules. This approach leads to simpler code and is preferable for platforms with a fixed set of peripheral devices, such as smartphones, where shipping drivers that support only the required devices enables smaller system image.

Specification reuse Our specification methodology ensures mutual independence of device and OS specifications, and thus facilitates their reuse. We have not yet carried out a substantial evaluation of such reuse; however we report our limited experience based on synthesizing two SPI drivers for the seL4 OS. The corresponding OS specification was initially developed during the work on the SPI driver for the exynos chipset. It was later used to synthesize a driver for the STM32F10 chipset. We were able to reuse most of the original specification. Minor changes (8 lines of code) were required in the part of the specification describing configuration functionality of the driver, since the STM SPI controller supports a number of ad hoc transfer modes. We expect to observe similar pattern for other devices and operating systems: generic OS specifications can be reused with localized, device-specific changes required to support non-standard device features.

Performance of synthesized drivers Our synthesized drivers implement effectively identical device control logic to their conventional counterparts and therefore have similar performance. We benchmarked the USB webcam driver, which is the most performance-critical one among our case studies. We measured CPU load and data throughput generated by the conventional and synthesized drivers for varying bitrates. We obtained identical results, modulo measurement errors, for both drivers in all cases.

8 Related work

Device driver reliability has been an active area of research for a number of years. Some of the techniques for dealing with buggy drivers include runtime isolation [27, 17], virtualisation [18], static verification [2, 9, 21], symbolic execution [7], language-based protection [34, 23], domain-specific languages [11, 19], hardware-software co-verification [25], etc.

This research has demonstrated the effectiveness of formal techniques in improving driver reliability. Interestingly, formal approaches to driver correctness fall into methods that verify existing drivers and methods that combine verification with an improved driver architecture. The latter rely on language and architectural support to eliminate entire families of driver bugs *by design*. Recent examples include the P programming language [11] and the active driver framework [1], which facilitate the development and automatic verification of asynchronous event-driven code. Our work can be seen as taking this correctness-by-construction approach to the extreme by generating drivers in an automated fashion.

9 Conclusion and future work

We presented the design and implementation of the Termite driver synthesis tool. Termite is the first tool to marry automatic game-based synthesis with conventional manual development. It is also the first practical synthesis tool based on abstraction refinement. Finally, it is the first synthesis tool to support automated debugging of input specifications.

Based on our experimental results, we consider Termite to be an important step towards truly practical device driver synthesis. In particular, our synthesis algorithm is able to efficiently handle real-world device specifications, while the user-guided approach reliably leads to high-quality code.

Our ongoing research focuses on solving the key remaining problems described in Section 6, primarily the DMA problem, which poses the main obstacle to synthesis of more complex drivers, and the grey-box synthesis problem, which limits the degree of automation achieved by Termite. Next, we will explore ways to improve the quality of automatically generated code and thus further reduce the need for user involvement. This includes performance- and power-aware synthesis. Finally, we plan to investigate automatic synthesis of hardened device drivers, i.e., drivers that gracefully handle misbehaving devices [15].

References

[1] S. Amani, P. Chubb, A. Donaldson, A. Legg, K. C. Ong, L. Ryzhyk, and Y. Zhu. Automatic verification of active device drivers. *ACM Operating Systems Review*, 48(1), May 2014.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *1st EuroSys Conference*, pages 73–85, Leuven, Belgium, Apr. 2006.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[4] L. Cai and D. Gajski. Transaction level modeling: an overview. In *"1st International Conference on Hardware/Software Codesign and System Synthesis"*, pages 19–24, Newport Beach, CA, USA, 2003.

[5] P. Cerny, T. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV*, Saint Petersburg, Russia, July 2013.

[6] P. Cerny, T. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Regression-free synthesis for concurrency. In *CAV*, Vienna, Austria, July 2014.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1):2:1–2:49, Feb. 2012.

[8] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *18th ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alta, Canada, Oct. 2001.

[9] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[10] L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. In *18th International Conference on Concurrency Theory*, pages 74–89, Lisboa, Portugal, Sept. 2007.

[11] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 321–332, Seattle, Washington, USA, 2013.

[12] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 191–202, Portland, Oregon, 2002.

[13] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *20th USENIX Large Installation System Administration Conference*, pages 101–111, Washington, DC, USA, 2006.

[14] Intel Corporation. Cofluent technolofy. `http://www.intel.com/content/www/us/en/cofluent/cofluent-difference.html`.

[15] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, 2009.

[16] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct. 2009.

[17] B. Leslie, P. Chubb, N. FitzRoy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sept. 2005.

[18] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, USA, Dec. 2004.

[19] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *4th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, CA, USA, Oct. 2000.

[20] 16550 UART core. `http://opencores.org/project,a_vhd_16550_uart`.

[21] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: ten years later. In *16th International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 305–318, Newport Beach, CA, USA, Mar. 2011.

[22] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380, Jan. 2006.

[23] M. J. Renzelmann and M. M. Swift. Decaf: Moving device drivers to a moderm language. In *USENIX Annual Technical Conference*, San Diego, CA, USA, June 2009.

[24] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct. 2009.

[25] L. Ryzhyk, J. Keys, B. Mirla, A. Raghunath, M. Vij, and G. Heiser. Improved device driver reliability through hardware verification reuse. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, Mar. 2011.

[26] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the starting problem: device drivers as self-describing artifacts. In *1st EuroSys Conference*, pages 45–57, Leuven, Belgium, 2006.

[27] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *19th ACM Symposium on Operating Systems Principles*, Bolton Landing (Lake George), New York, USA, Oct. 2003.

[28] Synopsys. Virtual prototyping models. `http://www.synopsys.com/Systems/VirtualPrototyping/VPModels`.

[29] W. Thomas. On the synthesis of strategies in infinite games. In *12th Annual Symposium on Theoretical Aspects of Computer Science*, pages 1–13, 1995.

[30] A. Walker and L. Ryzhyk. Predicate abstraction for reactive synthesis. In *FMCAD*, Lausanne, Switzerland, Oct. 2014.

[31] Wind River. Wind River Simics Model Builder user guide. version 4.4, Sept. 2010.

[32] WindRiver Simics DS12887 Model. `http://www.windriver.com/products/simics`.

[33] R. Yavatkar. Era of SoCs, presentation at the Intel Workshop on Device Driver Reliability, Modeling and Synthesis, Mar. 2012.

[34] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th USENIX Symposium on Operating Systems Design and Implementation*, pages 45–60, Seattle, WA, USA, Nov. 2006.