



PRINCETON UNIVERSITY

Department of Computer Science, 35 Olden Street, Princeton, NJ 08540

Phone: (609) 751-2222 Fax: (609) 258-1771

Email: abadam@cs.princeton.edu

Cover Letter

Anirudh Badam

I am looking for a **full-time researcher** position or a **tenure-track** faculty position at the assistant professor level in **Computer Science**. I am currently completing a Ph.D. at the Department of Computer Science, Princeton University, in the area of **computer systems** with focus on memory management within **operating systems**. My dissertation is entitled "On Bridging the Performance and Structural Gap Between Memory and Storage Using New Non-volatile Memory Technologies." Conceptually, my research aims to simplify the development of large scale data-intensive applications by providing them with better operating system support for new non-volatile memory technologies like NAND-flash and phase change memory.

My detailed C.V. is attached to this letter. I have attached a statement of research interests describing my current research and some future research directions that I wish to explore. Additionally, I have also attached a statement of teaching philosophy that provides a description of what my aim as a teacher would be. Finally, I have attached two of my publications. The following five people have agreed to serve as my references:

1. Dr. Vivek S. Pai, Associate Professor of Computer Science at Princeton (my thesis adviser)
2. Dr. Michael J. Freedman, Assistant Professor of Computer Science at Princeton (one of my thesis committee members)
3. Dr. Konstantina Papagiannaki, Head of the Internet Systems and Networking Group at Telefonica Research (a prior internship mentor)
4. Dr. Michael Kaminsky, Adjunct Research Scientist at the Intel Science and Technology Center for Cloud Computing (a prior internship mentor)
5. Dr. Robert Dondero, Lecturer of Computer Science at Princeton (I was his teaching assistant in the past)

You can obtain more information about me from my website:
<http://www.cs.princeton.edu/~abadam>

I thank you very much for your kind consideration.

Sincerely,
Anirudh Badam

Anirudh Badam

Dept. of Computer Science
35 Olden St.
Princeton, NJ 08540

Cell: 609-751-2222 & Fax: 609-258-1771
Email: abadam@cs.princeton.edu
<http://www.cs.princeton.edu/~abadam>

Education

Princeton University

Ph.D. in Computer Science Expected May 2012
M.A. in Computer Science April 2008
Advisor: Vivek S. Pai

Indian Institute of Technology, Madras

B.Tech in Computer Science June 2006
Minor in Operations Research

Honors

Siebel Scholar, Class of 2012 – awarded annually for academic excellence and demonstrated leadership to 85 top students from the worlds leading graduate schools.

Wu Prize for Excellence 2010 – awarded to Princeton University’s engineering graduate students who have performed at the highest level as scholars and researchers.

HashCache, a research project, named as one of the Top 10 Emerging Technologies of 2009 by MIT’s TechReview Magazine.

Technology for Developing Regions Fellowship, Princeton University, 2008.

Best Intern Award 2005 for Project Bluestreak, Extreme Blue Internship, IBM.

Merit Award from IIT Madras for being among the top 30 freshmen in the institute.

Ranked 5th and 57th among 200,000 students who appeared in the preliminary and main rounds, respectively, of the Indian Institute of Technology Joint Entrance Examination of 2002.

Prathiba Scholarship for excellence in high school, Govt. of India, 2002.

NCERT National Talent Search Examination Scholarship, Govt. of India, 2000.

Selected Publications

Anirudh Badam, Vivek S. Pai, “SSDAlloc: Hybrid RAM/SSD Memory Management Made Easy”, In Proc. USENIX NSDI ’11, Boston, MA, March 2011.

Anirudh Badam, Kyoungsoo Park, Vivek S. Pai, Larry L. Peterson, “HashCache: Cache Storage for the Next Billion”, In Proc. USENIX NSDI ’09, Boston, MA, April 2009.

Anirudh Badam, David W. Nellans, Vivek S. Pai, “TEAM: Transparent Expansion of Application Memory”, In Submission.

Anirudh Badam, Vivek S. Pai, “Chameleon: Better Non-volatile Memory Support via Shapeshifting Virtual Memory Pages”, In Submission.

Anirudh Badam, David W. Nellans, Vivek S. Pai, “Application Driven Flash Translation Layers”, In Submission.

Anirudh Badam, Dongsu Han, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, Srinivasan Seshan, “The Hare and the Tortoise: Taming Wireless Losses by Exploiting Wired Reliability”, In Proc. ACM MobiHoc ’11, Paris, France, May 2011.

Anirudh Badam, Bheemarjun Reddy Tamma, Sivaram M. C. “K-Tree: A multiple tree multimedia multicast protocol”, In Proc. HiPC 2006, Bangalore, India, December 2006.

Bheemarjun Reddy Tamma, **Anirudh Badam**, Sivaram M. C., Ramesh Rao, “K-Tree: A multiple tree multimedia multicast protocol”, Computer Networks 54(11): 1864-1884 (2010).

Selected Research Projects

HashCache

October 2006 - March 2009

Princeton University

Princeton, NJ

Description: HashCache involves the development of an efficient cache indexing mechanism. We have developed a new cache indexing mechanism for caches which consumes 6–20x less memory compared to the state-of-the-art indexing mechanisms. Our research directly implies that larger amounts of data can be indexed with the same amount of memory when compared to the present day high performance indexing mechanisms. This provides benefits for high-performance HTTP proxies, network accelerators that cache packets, data deduplicators that need to index arbitrary chunks of data and object caches that cache the results of SQL queries to speed up database operations.

SSDAlloc

April 2009 - March 2011

Princeton University

Princeton, NJ

Description: SSDAlloc is an easy to use hybrid SSD/RAM object based virtual memory management tool in combination with a runtime that transparently tiers objects between DRAM and flash memory. It helps programmers build high-performance flash memory based applications (comparable to that of application rewrite) while requiring very few modifications to application code – restricted only to the memory allocation portions of the program. Compared to existing transparent DRAM and flash memory tiering mechanisms, SSDAlloc increases the performance by 4–15x. Additionally, it can increase the lifetime of the flash memory device by 32x for a given workload over existing techniques.

fio_malloc

February 2011 -

Fusion-io Inc.

Salt Lake City, UT

Description: fio_malloc is an ongoing collaboration between Fusion-io and Princeton University to use enterprise class flash memory as a DRAM substitute in the virtual memory hierarchy. DRAM has a high initial cost and a high power requirement. Therefore, applications with high throughput requirements often have high capital requirements. In this project, we aim to address the impediments inside traditional operating systems to using flash memory as high-performance swap device. Additionally, we also explore techniques to optimize flash memory management by exposing the flash translation layer directly to the OS and the application. As a result, flash memory can now be managed in workload centric ways.

Chameleon

October 2011 -

Princeton University

Princeton, NJ

Description: Chameleon is a virtual memory management mechanism that performs cross-boundary optimizations between the application, the library and the OS to provide better support for new non-volatile memories. It reorganizes virtual memory so that applications can obtain all the desirable properties from a tiering system: It provides transparency by operating via virtual memory, it provides page-fault free access to all of DRAM via virtual memory and it optimizes the usage of non-volatile memory by allowing applications to localize reads and writes at an object level as opposed to page level.

HTPPR

June 2009 - May 2011

Intel Labs

Pittsburgh, PA

Description: HTPPR aims to develop a multi-link transport protocol primarily aimed at settings where a wireless link is present along with a wired link between the nodes. Such hybrid links can be found in settings like home networks, wireless LANs inside datacenters, and long distance WiFi links that are accompanied by low bandwidth PPPoE links. HTPPR obtains better bandwidth on the lossy wireless link by leveraging the reliability of the wired link in a bandwidth efficient manner.

Reliable Video Multicast

September 2005 to May 2006

HPCN Lab, IIT Madras

Chennai, India

Description: The work involved the exploration of multiple multicast trees for reliable transmission of video in an ad-hoc wireless network. Apart from formally proving that finding the maximum number of node disjoint trees is a hard problem, we developed a competitive online algorithm for creating multiple maximally node disjoint trees which paved the way for a reliable multiple-tree multicast mechanism.

Selected Talks and Presentations

“HashCache: Cache Storage for the Next Billion”:

1. Master’s Thesis Defense, Princeton, NJ, Jan 2008.
2. Conference talk at USENIX NSDI, Boston, MA, May 2009.

“SSDAlloc: Hybrid RAM/SSD Memory Management Made Easy”:

3. Work-in-progress talk at USENIX ATC 2010, Boston, MA, Jun 2010.
4. Invited talk at NetAPP Inc., Boston, MA, Jun 2010.
5. Invited talk at IBM T.J.Watson, Hawthorne, NY, Jul 2010.
6. Workshop talk at IBM T.J.Watson Nonvolatile Memory Workshop, Hawthorne, NY, Sep 2010.
7. Invited talk at Fusion-io Inc., Salt Lake City, UT, Feb 2011.
8. Workshop talk at UCSD NVMW 2011, San Diego, CA, Mar 2011.
9. Conference talk at USENIX NSDI 2011, Boston, MA, Apr 2011.

“On Bridging the Performance and Structural Gap Between Memory Storage Layers Using New Non-volatile Memory Technologies”:

10. PhD Thesis Proposal, Princeton, NJ, Nov 2010.

“HTPPR: Tackling Wireless Losses by Exploiting Wired Reliability”:

11. Invited talk at Intel Labs Pittsburgh Open House, Pittsburgh, PA, Nov 2009.
12. Conference talk at ACM MobiHoc, Paris, France, May 2011.

Professional Experience

Research Intern at Fusion-io Inc. , Salt Lake City, Utah	February 2011 - August 2011
Research Intern at Intel Labs Pittsburgh	June 2009 - August 2009
Research Intern at HP Labs Princeton	June 2008 - August 2008
Teaching Assistant for Advanced Programming Techniques	Spring Semester 2008
Teaching Assistant for Introduction to Programming Systems	Fall Semester 2007
Research Intern at IBM Research Lab , India	May 2005 - July 2005

Patents

Larry Peterson, Vivek S. Pai, Sunghwan Ihm, **Anirudh Badam**, KyoungSoo Park. Systems and Methods for Network Acceleration and Efficient Indexing for Caching Filesystems: Final approval pending. US provisional patent was filed on March 10th 2010.

Anirudh Badam, Vivek S. Pai. SSDAlloc: Hybrid SSD/DRAM Memory Management Made Easy. US provisional patent was filed on March 15th 2011.

References

Vivek S. Pai (thesis adviser)

Associate Professor, Department of Computer Science, Princeton University

Email: vivek@cs.princeton.edu

Michael J. Freedman (thesis committee member)

Assistant Professor, Department of Computer Science, Princeton University

Email: mfreed@cs.princeton.edu

Konstantina Papagiannaki (prior internship mentor)

Head of the Internet Systems and Networking Group at Telefonica Research, Spain

Email: dina@tid.es

Michael Kaminsky (prior internship mentor)

Adjunct Research Scientist at Intel Science and Technology Center for Cloud Computing

Email: michael.e.kaminsky@intel.com

Robert Dondero (I was his teaching assistant in the past)

Lecturer, Department of Computer Science, Princeton University

Email: rdondero@cs.princeton.edu

Statement of Research Directions

Anirudh Badam

Advancements in computer, electrical and material sciences are taking us closer every day to a computer that has a single, large, persistent, fast and directly addressable memory unit. Today, however, one could develop novel software techniques to bridge the performance, and structural gap between memory and storage layers. This will help us move one step closer to such a memory unit. Conceptually, my research aims to use non-volatile memory technologies like NAND-flash memory (flash) for this purpose.

Today, a combination of virtual memory, local and network based filesystems are used to build large data-centric systems [3, 13, 22, 33, 34]. These systems expose a single convenient namespace that can be used across an entire cluster for the applications' convenience. However, each of them has to deal with the overhead of having to tier data between memory and storage in a custom manner.

New technologies like flash can help thin the gap between memory and storage, and help reduce the overhead of developing such applications. These new technologies can be thought of as slower but persistent random access memories or as incredibly fast block storage devices. Today, flash can scale to 20TB within a single rack unit (RU) server [19]. Additionally, state-of-the-art flash consumes ten times less power when compared to high-density DRAM. This allows for fundamentally more energy efficient datacenters that can process larger amounts of data faster than datacenters that are purely DRAM and disk based.

The integration of these technologies into the memory hierarchy will not be without challenges. These technologies are neither as transparent as CPU caches nor as low in latency as DRAM. They will make us rethink the way we design and program applications like the way CPU caches did. Moreover, since their behavior slots somewhere between DRAM and magnetic disks, the ways to target them can range from application approaches, programming techniques to OS avenues. In particular, I think the following questions are of interest in the near future:

1. *The Performance Problem:* How must virtual memory be managed so that multiple underlying physical memory technologies with different performance and behavioral characteristics can coexist?
2. *The Distribution Problem:* How must the operating system change such that software based tiering of data scales well across multiple threads and processes, and multiple cores and machines?
3. *The Structuring Problem:* How must virtual memory be managed such that memory management provides the structuring and reliability of data similar to what filesystems and object stores provide?
4. *The Strategic Problem:* What additional interfaces must be introduced so that programmers can design applications that are “tiering/multicore/network-aware” when there is scope for improving latency?

Addressing these problems can reduce the complexity of building large data-centric systems – programmers can focus purely on application logic. Personally, I follow research in the fields of programming languages, database management systems, operating systems and computer architecture working towards this goal with utmost interest and fascination. My previous and present work can be seen as an attempt towards realizing this goal. I aspire to continue to work in this direction in the future.

Current Work

This section highlights how my current work focuses on addressing the software tiering issues in network servers since many users are directly impacted by their performance.

HashCache: Reopening the Research on Efficient and Scalable Indexing

The design of indexes for caches that power web-scale systems [1, 11, 12, 31, 17, 35] had stabilized in the 1995-2000 timeframe when activity on web caches was at its peak. Moreover, Moore's Law ensured that the available memory for storing indexes doubled at a regular interval. However, the sudden explosion of data caused by the rise in social networking and people moving more of their data into the cloud, warranted a reopening of the research for more efficient indexing mechanisms. In HashCache [9], I developed an indexing mechanism that required 6–20x less memory compared to the then state-of-the-art cache indexing techniques.

HashCache’s efficient indexing mechanism allows web-scale caches to have their entire index cost-effectively in faster memory technologies like DRAM or flash. This allows the cache to respond to membership queries at a low-latency. In particular, the motivation behind HashCache was to provide large-scale and affordable web caches to regions with poor Internet connectivity. Huge caches make much more sense for such regions with massive bandwidth costs.

HashCache improves the indexing efficiency such that large caches can be installed on commodity servers. For example, a 1TB cache containing 1KB objects in a cache like Squid [37] would require ~ 160 GB of space just for the index. Without that much DRAM or flash, a portion of that index would overflow to disk. While the added latency of accessing disk might be condonable in case of a cache hit, it is really a nuisance for a cache miss. HashCache requires only 8GB of memory for indexing such a cache, a space that most commodity servers can afford. Recently, such efficient indexing motivated other researchers to examine the problem in various related domains [2, 3, 15, 28].

For larger caches with much smaller objects, usually needed for network accelerators [25] and data deduplicators [14], I proposed using flash to augment DRAM to store the index. I optimized the techniques in HashCache such that they work better when used over flash by using a flash based custom object store [7]. While the custom interfaces to accessing flash that are proposed in various systems [2, 3, 7, 10, 38] make flash more accessible to programmers, a more transparent approach, purely via virtual memory, is needed for a widespread adoption of flash by programmers.

SSDAlloc: Bringing Flash Closer to the Application

SSDAlloc [8] rethinks flash in virtual memory in an application centric way to get higher performance and lifetime out of a flash memory device. It not only moves flash closer to the application by exposing it via virtual memory but also improves performance over existing transparent mechanisms by 4–15x. Additionally, by using SSDAlloc, applications can get up to 32 times more life out of a flash memory device for a given workload over existing transparent mechanisms.

Unlike DRAM or disk, flash memory’s blocks can be written to only after they are first “erased”. These blocks are also much larger (>128 KB) than a page size. Additionally, each of these blocks can be erased only a limited number of times before it becomes unusable. While flash translation layers [23] expose a clean small-block (<8 KB) interface to the OS by masking flash’s limitations, memory-intensive applications are often modified to work well with flash [16, 18, 26, 30] by using flash based log-structured object stores. Being able to restrict writes to an object (where objects are smaller than blocks) helps these applications obtain better performance and higher lifetime from flash.

Existing transparent techniques, swap [21] and systems that improve swap’s performance [27, 29, 32, 36] cannot not provide such a write localization: writes via virtual memory can be tracked only at the granularity of the size of a virtual memory page and no less without having to perform an expensive diff operation.

SSDAlloc is a memory allocator in combination with a runtime mechanism that manages DRAM and flash for an application. SSDAlloc reorganizes the way virtual memory is used by an application so that it can obtain information that can localize writes. For each object that an application allocates via SSDAlloc, an entire virtual memory page is given to the application. The object is placed at the beginning of the virtual memory page and the rest of the space in that page is not used for allocation. By using virtual memory page level usage information provided by hardware, the OS can precisely track the object that has been written.

Flash memory device is managed as a log-structured object store as opposed to a page store so that only modified objects are written to flash. By a clever page cache design, SSDAlloc avoids the physical memory space wastage that could result from such frivolous usage of virtual memory. SSDAlloc uses the rest of the physical space in each virtual memory page that is resident in DRAM to cache other objects to speed up operations to the object store on the flash memory device. SSDAlloc’s runtime transparently works with the virtual memory protection mechanism to create and destroy such pages transparent to the application.

Bridging the Gap Between the Application, the Library and the OS

In my next work, I propose bridging the gap further between the application, the library and OS. SSDAlloc not only provides transparency but it also localizes writes to objects. However, in SSDAlloc, only the objects at the beginning of each physical DRAM page can be accessed without runtime intervention. All other objects (cached in DRAM at a different location than its virtual memory page or stored currently on flash) are brought in to core via runtime intervention because their virtual memory pages are locked. Such a

gratuitous runtime involvement can be a nuisance for applications with a working set that can fit in DRAM.

In Chameleon [6], I propose a solution that has no such overhead. It is a memory allocation tool in combination with an OS supported runtime that decouples virtual memory management from the OS’s physical memory management policies. The goal of Chameleon’s virtual memory management is to let applications localize accesses to objects. Inside the OS, its DRAM management goal is to provide access to all the data in DRAM without runtime intervention and object based management of flash.

Chameleon can be summarized using the following simple example: An application can be allocated the first two KB of a virtual memory page and the last two KB of another virtual memory page. Since the mapped/allocated portions of these virtual memory pages do not intersect, they can share a single physical memory page while resident (pages are marked as dirty at the virtual memory level). This not only localizes the application’s writes to 2KB regions instead of 4KB pages but it also provides unhindered free access to the entire 4KB physical memory page. Chameleon’s memory allocator generalizes this idea to arbitrarily sized sparse mappings. Its runtime extension to the OS takes care of DRAM and flash management independently of how virtual memory is allocated.

Application Centric Ways to Share and Schedule Resources

It is critical for the OS to perform the necessary steps of a replacement policy without much CPU overhead. Traditional OSes are not designed this way because their page-out systems were originally designed for slow disks where swapping was considered worst-case behavior; flash’s low access latency warrants faster techniques.

Unlike existing operating systems, in TEAM [5], we present a lightweight way to keeping track of virtual memory centric usage statistics (LRU information) as opposed to maintaining system-wide physical memory page usage statistics. Physical memory statistics force an OS to perform CPU-intensive scans of virtual memory regions of processes to pick a page out candidate [21].

Increasing the flash backed memory would mean that the problem of custom memory management via interfaces like `madvise` is that much harder. While TEAM allows the OS to make more virtual memory specific decisions, it still does not provide better tools to perform custom memory management.

Providing a “good-enough” hint as opposed to a precise one might not be harmful simply because flash is two orders of magnitude faster than disk. Moreover, with large amounts of flash there might be many “good-enough” candidates. In `Madvise+` [4], we propose new techniques to simplify custom memory management where applications can quickly identify many “good-enough” candidates for paging-out and prefetching while the OS might take more time to figure these things out by itself.

Future Research Directions

The overall aim of my future research is to simplify the development of data-centric and memory-intensive applications [3, 13, 22, 33, 34] and help them scale by removing the impediments to the accessibility of new memory technologies. The future is going to present more opportunities as these data-centric applications become more widely deployed.

While my current research focuses on the *performance problem* and the *strategic problem*, it barely scratches the surfaces on the *distribution problem* and the *structural problem*. It is important for a transparent tiering system to help data-centric applications scale across multiple cores and multiple machines.

Scaling the application centric and transparent tiering of data to multiple cores can present some interesting challenges with respect to scalable datastructure design and resource scheduling within the OS. Scaling such a view of data across multiple machines within a network is also going to present additional challenges. These are not restricted to ownership (as to which machine within the cluster owns what data) or consistency models.

At the same time, it is also important for these applications to obtain storage system like reliability, structuring and manageability directly from a transparent tiering system as opposed to how it happens in OSes today – memory and storage can be tiered but have to be managed separately. Today, the mechanisms (application/library/runtime/OS level) that tier memory transparently are far from providing the functionalities that are provided by filesystems and object stores.

Below, I elaborate on how these problems are important and interesting, and must be addressed in the near future. In all the problems, my aim will be to bring as much support for large-scale data-centric applications from within a tiered system as possible.

Bridging the Structural Gap Between Memory and Storage

Data-centric computing techniques provide a custom interface to data that exposes a single convenient namespace to the user. However, they tier data underneath between various memory technologies (DRAM, flash, and/or disk). Better support can be provided for such applications if libraries, runtime and OS extensions that tier memory transparently can provide at least a few of these capabilities scalably.

Today, memory is used as a “volatile cache” and application data is persisted to a filesystem or an object store. New memory technologies that are not only fast but also persistent can be leveraged to bridge this behavioral gap between memory and storage and reduce the programming overhead. Clearly, regardless of the approach chosen, knowledge and wisdom of filesystems and object stores must be invoked in solving this problem because these interfaces have been chosen to help such applications scale [3, 20, 24, 33].

My current work on HashCache [9], SSDAlloc [8] and Chameleon [6] provide better support for object stores in a transparent tiering system. I wish to continue along those lines to provide better support for object stores inside transparently tiered systems. However, virtual memory management tools can definitely afford more direct filesystem support than basic file mapping operations. In particular, I think the following problems will be of interest:

1. Filesystems have a great advantage in terms of the name space and the directory tree to provide a clear structuring that helps create, share and use files across the entire OS. How must memory be managed such that it provides similar capabilities to applications?
2. Filesystems simplify the problems of reliability, security and management by abstracting themselves cleanly from applications. How can physical memory be abstracted away from the flat virtual memory address space that the applications use to provide similar properties?

In both the cases, I wish to explore new techniques like the ones I used in SSDAlloc and Chameleon: virtual memory is decoupled from physical memory with the help of a clever memory allocator and some OS extensions. Similarly, one must leverage the low access latency and persistence of new memory technologies.

Tiering Data for a Multicore System

The datastructures managing the virtual memory allocation, ones that manage the DRAM and the non-volatile memory device and ones that manage the page tables must all be built in a scalably parallelizable fashion. This is a daunting task considering the fact that tiering is now a software task and happens inside the OS. Unlike disk, new memory technologies have inherent parallelism that makes the task harder.

Various techniques ranging from lock-less datastructures, fine grained locking to message passing between synchronizing CPUs have to be explored. Overall, the aim should be to build all these datastructures in a way that reduces software intervention and locking as much as possible.

Tiering Data Over the Network

Tiering data transparently across a network can help distributed data-centric systems [3, 13, 22, 33, 34] scale better. In particular, I am interested in exploring various data models so that inexpensive and energy-efficient non-volatile memory technologies can be used as caches to hedge against network access when possible.

In particular, I wish to explore three different kinds of data models. First, some nodes in the network can be thought of as the owners of the data, while the rest are clients accessing and caching the data via the tiering system (SAN model). For the second model, I wish to explore a more decentralized approach to data ownership (MapReduce model). Finally, an intermediate model can also be explored where there is a mix of both – some nodes having more appetite for data than others.

Synchronization of data across these machines becomes a very interesting problem considering the fact that computer networks are slower compared to CPU interconnects. The tiering system has to take this latency into account while providing solutions. Additionally, various systems can require various kinds of consistency models. Since I propose that the tiering system provide some structuring for the data, such structure can potentially be maintained in a consistent manner across the network of machines. This can potentially reduce the burden of applications having to maintain consistency in a tiered system.

Systems like Madvise+ [4] can be modified such that various regions of the virtual memory can be managed separately with various kinds of policies for data synchronization. However, this is under the assumption that the virtual memory manager understands how the data is structured. Hence, this is a problem that is closely related to the one that tries to bridge the gap between virtual memory and filesystems.

References

- [1] Akamai Inc. <http://www.akamai.com>.
- [2] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *Proc. 7th USENIX NSDI*, San Jose, CA, Apr. 2010.
- [3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proc. 22nd ACM SOSP*, Big Sky, MT, Oct. 2009.
- [4] A. Badam, D. W. Nellans, and V. S. Pai. Application Driven Flash Translation Layers. In *Submission*.
- [5] A. Badam, D. W. Nellans, and V. S. Pai. TEAM: Transparent Expansion of Application Memory. In *Submission*.
- [6] A. Badam and V. S. Pai. Chameleon: Better Non-volatile Memory Support via Shapeshifting Virtual Memory Pages. In *Submission*.
- [7] A. Badam and V. S. Pai. Beating Netbooks into Servers: Making Some Computers More Equal Than Others. In *Proc. 3rd ACM NSDR*, Big Sky, MT, 2009.
- [8] A. Badam and V. S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proc. 8th USENIX NSDI*, Boston, MA, Mar. 2011.
- [9] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proc. 6th USENIX NSDI*, Boston, MA, Apr. 2009.
- [10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe With Next-Generation, Non-Volatile Memories. In *Proc. ACM ASPLOS*, Newport Beach, CA, Mar. 2011.
- [11] CoDeeN: A Content Distribution Network for PlanetLab. <http://codeen.cs.princeton.edu/>.
- [12] Coral: The Coral Content Distribution Network. <http://www.coralcdn.org>.
- [13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th USENIX OSDI*, Dec. 2004.
- [14] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proc. USENIX ATC*, Boston, MA, June 2010.
- [15] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash. In *Proc 30th ACM SIGMOD*, Athens, Greece, June 2011.
- [16] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS Buffer Pool Using SSDs. In *Proc 30th ACM SIGMOD*, Athens, Greece, June 2011.
- [17] Facebook Engineering. https://www.facebook.com/note.php?note_id=39391378919.
- [18] FAWN Energy Efficient Sort. <http://sortbenchmark.org/>.
- [19] Fusion-io: ioDrive Octal. <http://www.fusionio.com/platforms/iodrive-octal/>.
- [20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google Filesystem. In *Proc. 19th ACM SOSP*, Lake George, NY, Oct. 2003.
- [21] M. Gormen. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.
- [22] Graph500 Graph Traversal Benchmark. <http://www.graph500.org/index.html>.
- [23] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proc. ACM ASPLOS*, Washington, DC, Mar. 2009.
- [24] Hadoop Filesystem. <http://hadoop.apache.org/hdfs/>.
- [25] S. Ihm, K. Park, and V. S. Pai. Wide-area Network Acceleration for the Developing World. In *Proc. USENIX ATC*, Boston, MA, June 2010.
- [26] T. Kgil and T. N. Mudge. Flashcache: A NAND Flash Memory File Cache for Low Power Web Servers. In *Proc. of CASES*, 2006.
- [27] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A New Linux Swap System for Flash Memory Storage Devices. In *Proc. of ICCSA*, June 2008.
- [28] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. 23rd ACM SOSP*, Cascias, Portugal, Oct. 2011.
- [29] M. Lin, S. Chen, G. Lu, and Z. Zhou. Optimizing Linux Swap System for Flash Memory. In *IEEE Electronic Letters*, 2011.
- [30] LLNL Graph Traversal Infrastructure. <http://www.fusionio.com/blog/llnl-leverages-iomemory-to-process-68-billion-node-graph/>.
- [31] Memcache. <http://www.danga.com/memcached>.
- [32] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating System Support for NVM+DRAM Hybrid Main Memory. In *Proc. 11th USENIX HotOS*, Monte Verita, Switzerland, May 2009.
- [33] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. 23rd ACM SOSP*, Cascias, Portugal, Oct. 2011.
- [34] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, Oct. 2010.
- [35] Riverbed. <http://www.riverbed.com/us>.
- [36] M. Saxena and M. M. Swift. FlashVM: Virtual Memory Management on Flash. In *Proc. USENIX ATC*, Boston, MA, June 2010.
- [37] Squid HTTP Cache. <http://www.squid-cache.org/>.
- [38] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proc. ACM ASPLOS*, Newport Beach, CA, Mar. 2011.

Statement of Teaching Philosophy

Anirudh Badam

My aim as a teacher is to design new undergraduate and graduate courses, to guide graduate students and to help motivated undergraduate students grasp the basics of research in my field of interest that is computer networks and systems.

My first professional teaching experience was when I was a senior at my undergraduate institution, Indian Institute of Technology, Madras. I was assigned the task of a lab assistant who would help freshman undergraduates with basic C and UNIX programming skills. I thoroughly enjoyed interacting with the students. I used to have frequent discussions and debates with them to help them analyze why certain aspects of C and UNIX were designed the way they were. I believe that these discussions helped me form the basis of my teaching philosophy – that students need an environment where they can interactively dissect, critique and evaluate the design of systems to learn faster.

As a second year graduate student at Princeton, I was fortunate enough to be assigned as a teaching assistant to two intensive programming courses: Introduction to Programming Systems ¹ and Advanced Programming Techniques ². The experience that I obtained from teaching these courses helped me further concretize my teaching philosophy.

Inculcating the habit of grasping a concept not just by understanding the best available solution but also by critiquing the alternatives was fruitful. I wanted them to understand that programming was ultimately just a methodology to solve problems and often there are tradeoffs that one has to make. One of the most interesting discussions that I had with the students was about the advantages and disadvantages of using the standard C library functions via two interfaces: using shell commands vs writing new C application programs. While the passionate discussion fringed upon issues ranging from portability, software engineering effort to efficiency, I think ultimately the students benefited from the fact that they were able to dissect, critique and weigh various options to pick the right solution.

Keeping these experiences of mine in mind, I think that the five-point plan outlined as follows can help me make undergraduate and graduate level system design and implementation courses more rewarding on a general and personal level for each student.

1. **Keep the introductory programming courses interactive:** Programming is a field where peers can be motivating and can also be very good source of genuine insight. Learning programming is a very personal experience for many people where they need a right environment to learn as opposed to needing a presentation that states facts. These interactions can also be staged such that students can appreciate the work that was done before their time. Providing a “big picture” idea of system design interactively at the freshman and sophomore level could motivate more undergraduate students to develop a liking for programming.
2. **Personalize the content for more advanced programming courses:** Advanced programming courses designed around the needs of junior undergraduates can be personalized to enhance students’ design skills. Giving them the maximum amount of freedom with respect to choosing a programming assignment can achieve such a goal. Choice can also be given between, various platforms, frameworks, and programming environments. By giving them such a freedom, one can provide them with an early exposure to the life of a system designer; where programming is essentially the final phase and may be even the simplest phase if done right.
3. **Organize new kinds of programming contests:** One way to motivate young minds to go the extra mile to grasp better system design skills is by organizing innovative programming events. Programming contests can be designed to help the students gain a better understanding of new programming languages, compilers, computer networks, and operating systems unlike existing ones that focus mostly on abstract algorithms and datastructures.

¹<http://www.cs.princeton.edu/courses/archive/fall07/cos217/>

²<http://www.cs.princeton.edu/courses/archive/spring08/cos333/>

4. **Host an interactive blog about design and implementation of systems:** I wish to host a blog in which students and professors can write articles about anything pertaining to the design and implementation of new systems that they find interesting. It can be on any number of practical and theoretical challenges of realizing a particular idea; it can also be a critique about an existing system. The informal setting of a blog can also help students be more interactive and critical of each other's opinions. Such qualities are important for students wishing to design systems in the future.
5. **Organize a seminar to discuss latest academic work in system building:** The seminar would discuss the happenings in the academic world in a more formal setting when compared to a blog. Each seminar lecture can discuss one or two papers that were recently published and has aspects of novel system design and implementation. The course can be designed around the needs of an enthusiastic senior undergraduate student or a new graduate student who is looking for a broad exposure to the general area of systems.

In addition to teaching programming and system building courses interactively, I wish to take graduate students in my preferred field of research that is computer networks and systems. Here, I draw upon the wisdom that my advisor, Prof. Vivek S. Pai gave me. My goal as a graduate student advisor would simply be to pass on the same kind of support and fellowship that Vivek gave me. Vivek always told me that working on good problems is always the root to happiness in a graduate student's life – “work on high-impact problems and everything else will be fine”. Additionally, Vivek is a master of the art of customizing his guidance for each phase of a graduate student's life. He knew exactly when to give the maximum amount of freedom and when to closely evaluate progress. His timely insight and the willingness to give insane amounts of freedom always helped me make progress in a stress-free manner. I wish to emulate his style of guiding graduate students so that I can inspire them to work hard and work on important problems.

HashCache: Cache Storage for the Next Billion

Anirudh Badam*, KyoungSoo Park*⁺, Vivek S. Pai* and Larry L. Peterson*

**Department of Computer Science
Princeton University*

*+Department of Computer Science
University of Pittsburgh*

Abstract

We present HashCache, a configurable cache storage engine designed to meet the needs of cache storage in the developing world. With the advent of cheap commodity laptops geared for mass deployments, developing regions are poised to become major users of the Internet, and given the high cost of bandwidth in these parts of the world, they stand to gain significantly from network caching. However, current Web proxies are incapable of providing large storage capacities while using small resource footprints, a requirement for the integrated multi-purpose servers needed to effectively support developing-world deployments. HashCache presents a radical departure from the conventional wisdom in network cache design, and uses 6 to 20 times less memory than current techniques while still providing comparable or better performance. As such, HashCache can be deployed in configurations not attainable with current approaches, such as having multiple terabytes of external storage cache attached to low-powered machines. HashCache has been successfully deployed in two locations in Africa, and further deployments are in progress.

1 Introduction

Network caching has been used in a variety of contexts to reduce network latency and bandwidth consumption, ranging from FTP caching [31], Web caching [15, 4], redundant traffic elimination [20, 28, 29], and content distribution [1, 10, 26, 41]. All of these cases use local storage, typically disk-based, to reduce redundant data fetches over the network. Large enterprises and ISPs particularly benefit from network caches, since they can amortize their cost and management over larger user populations. Cache storage system design has been shaped by this class of users, leading to design decisions that favor first-world usage scenarios. For example, RAM consumption is proportional to disk size due to in-memory

indexing of on-disk data, which was developed when disk storage was relatively more expensive than it is now. However, because disk size has been growing faster than RAM sizes, it is now much cheaper to buy terabytes of disk than a machine capable of indexing that much storage, since most low-end servers have lower memory limits.

This disk/RAM linkage makes existing cache storage systems problematic for developing world use, where it may be very desirable to have terabytes of cheap storage (available for less than US \$100/TB) attached to cheap, low-power machines. However, if indexing a terabyte of storage requires 10 GB of RAM (typical for current proxy caches), then these deployments will require server-class machines, with their associated costs and infrastructure. Worse, this memory is dedicated for use by a single service, making it difficult to deploy consolidated multi-purpose servers. When low-cost laptops from the One Laptop Per Child project [22] or the Classmate from Intel [13] cost only US \$200 each, spending thousands of dollars per server may exceed the cost of laptops for an entire school.

This situation is especially unfortunate, since bandwidth in developing regions is often more expensive, both in relative and absolute currency, than it is in the US and Europe. Africa, for example, has poor terrestrial connectivity, and often uses satellite connectivity, back-hauled through Europe. One of our partners in Nigeria, for example, shares a 2 Mbps link, which costs \$5000 per month. Even the recently-planned “Google Satellite,” the O3b, is expected to drop the cost to only \$500/Mbps per month by 2010 [21]. With efficient cache storage, one can reduce the network connectivity expenses.

The goal of this project is to develop network cache stores designed for developing-world usage. In this paper, we present HashCache, a configurable storage system that implements flexible indexing policies, all of which are dramatically more efficient than traditional cache designs. The most radical policy uses no main

memory for indexing, and obtains performance comparable to traditional software solutions such as the Squid Web proxy cache. The highest performance policy performs equally with commercial cache appliances while using main-memory indexes that are only one-tenth their size. Between these policies are a range of distinct policies that trade memory consumption for performance suitable for a range of workloads in developing regions.

1.1 Rationale For a New Cache Store

HashCache is designed to serve the needs of developing-world environments, starting with classrooms but working toward backbone networks. In addition to good performance with low resource consumption, HashCache provides a number of additional benefits suitable for developing-world usage: (a) many HashCache policies can be tailored to use main memory in proportion to system activity, instead of cache size; (b) unlike commercial caching appliances, HashCache does not need to be the sole application running on the machine; (c) by simply choosing the appropriate indexing scheme, the same cache software can be configured as a low-resource end-user cache appropriate for small classrooms, as well as a high-performance backbone cache for higher levels of the network; (d) in its lowest-memory configurations, HashCache can run on laptop-class hardware attached to external multi-terabyte storage (via USB, for example), a scenario not even possible with existing designs; and (e) HashCache provides a flexible caching layer, allowing it to be used not only for Web proxies, but also for other cache-oriented storage systems.

A previous analysis of Web traffic in developing regions shows great potential for improving Web performance [8]. According to the study, kiosks in Ghana and Cambodia, with 10 to 15 users per day, have downloaded over 100 GB of data within a few months, involving 12 to 14 million URLs. The authors argue for the need for applications that can perform HTTP caching, chunk caching for large downloads and other forms of caching techniques to improve the Web performance. With the introduction of personal laptops into these areas, it is reasonable to expect even higher network traffic volumes.

Since HashCache can be shared by many applications and is not HTTP-specific, it avoids the problem of diminishing returns seen with large HTTP-only caches. HashCache can be used by both a Web proxy and a WAN accelerator, which stores pieces of network traffic to provide protocol-independent network compression. This combination allows the Web cache to store static Web content, and then use the WAN accelerator to reduce redundancy in dynamically-generated content, such as news sites, Wikipedia, or even locally-generated content, all of which may be marked uncacheable, but which tend to only change slowly over time. While modern Web

pages may be large, they tend to be composed of many small objects, such as dozens of small embedded images. These objects, along with tiny fragments of cached network traffic from a WAN accelerator, put pressure on traditional caching approaches using in-memory indexing.

A Web proxy running on a terabyte-sized HashCache can provide a large HTTP store, allowing us to not only cache a wide range of traffic, but also speculatively preload content during off-peak hours. Furthermore, this kind of system can be driven from a typical OLPC-class laptop, with only 256MB of total RAM. One such laptop can act as a cache server for the rest of the laptops in the deployment, eliminating the need for separate server-class hardware. In comparison, using current Web proxies, these laptops could only index 30GB of disk space.

The rest of this paper is structured as follows. Section 2 explains the current state of the art in network storage design. Section 3 explains the problem, explores a range of HashCache policies, and analyzes them. Section 4 describes our implementation of policies and the HashCache Web proxy. Section 5 presents the performance evaluation of the HashCache Web Proxy and compares it with Squid and a modern high-performance system with optimized indexing mechanisms. Section 6 describes the related work, Section 7 describes our current deployments, and Section 8 concludes with our future work.

2 Current State-of-the-Art

While typical Web proxies implement a number of features, such as HTTP protocol handling, connection management, DNS and in-memory object caching, their performance is generally dominated by their filesystem organization. As such, we focus on the filesystem component because it determines the overall performance of a proxy in terms of the peak request rate and object cacheability. With regard to filesystems, the two main optimizations employed by proxy servers are hashing and indexing objects by their URLs, and using raw disk to bypass filesystem inefficiencies. We discuss both of these aspects below.

The Harvest cache [4] introduced the design of storing objects by a hash of their URLs, and keeping an in-memory index of objects stored on disk. Typically, two levels of subdirectories were created, with the fan-out of each level configurable. The high-order bits of the hash were used to select the appropriate directories, and the file was ultimately named by the hash value. This approach not only provided a simple file organization, but it also allowed most queries for the presence of objects to be served from memory, instead of requiring disk access. The older CERN [15] proxy, by contrast, stored objects by creating directories that matched the components of the URL. By hashing the URL, Harvest was able to con-

System	Naming	Storage Management	Memory Management
CERN	URL	Regular Filesystem	Filesystem Data Structures
Harvest	Hash	Regular Filesystem	LRU, Filesystem Data Structures
Squid	Hash	Regular Filesystem	LRU & others
Commercial	Hash	Log	LRU

Table 1: System Entities for Web Caches

control both the depth and fan-out of the directories used to store objects. The CERN proxy, Harvest, and its descendant, Squid, all used the filesystems provided by the operating system, simplifying the proxy and eliminating the need for controlling the on-disk layout.

The next step in the evolution of proxy design was using raw disk and custom filesystems to eliminate multiple levels of directory traversals and disk head seeks associated with them. The in-memory index now stored the location on disk where the object was stored, eliminating the need for multiple seeks to find the start of the object.¹

The first block of the on-disk file typically includes extra metadata that is too big to be held in memory, such as the complete URL, full response headers, and location of subsequent parts of the object, if any, and is followed by the content fetched from the origin server. In order to fully utilize the disk writing throughput, those blocks are often maintained consecutively, using a technique similar to log-structured filesystem(LFS) [30]. Unlike LFS, which is expected to retain files until deleted by the user, cache filesystems can often perform disk cache replacement in LIFO order, even if other approaches are used for main memory cache replacement. Table 1 summarizes the object lookup and storage management of various proxy implementations that have been used to build Web caches.

The upper bound on the number of cacheable objects per proxy is a function of available disk cache and physical memory size. Attempting to use more memory than the machine's physical memory can be catastrophic for caches, since unpredictable page faults in the application can degrade performance to the point of unusability. When these applications run as a service at network access points, which is typically the case, all users then suffer extra latency when page faults occur.

The components of the in-memory index vary from system to system, but a representative configuration for a high-performance proxy is given in Table 2. Each entry has some object-specific information, such as its hash value and object size. It also has some disk-related

¹This information was previously available on the iMimic Networking Web site and the Volera Cache Web site, but both have disappeared. No citable references appear to exist

Entity	Memory per Object (Bytes)
Hash	4 - 20
LFS Offset	4
Size in Blocks	2
Log Generation	1
Disk Number	1
Bin Pointers	4
Chaining Pointers	8
LRU List Pointers	8
Total	32 - 48

Table 2: High Performance Cache - Memory Usage

information, such as the location on disk, which disk, and which generation of log, to avoid problems with log wrapping. The entries typically are stored in a chain per hash bin, and a doubly-linked LRU list across all index entries. Finally, to shorten hash bin traversals (and the associated TLB pressure), the number of hash bins is typically set to roughly the number of entries.

Using these fields and their sizes, the total consumption per index entry can be as low as 32 bytes per object, but given that the average Web object is roughly 8KB (where a page may have tens of objects), even 32 bytes per object represents an in-memory index storage that is 1/256 the size of the on-disk storage. With a more realistic index structure, which can include a larger hash value, expiration time, and other fields, the index entry can be well over 80 bytes (as in the case of Squid), causing the in-memory index to exceed 1% of the on-disk storage size. With a single 1TB drive, the in-memory index alone would be over 10GB. Increasing performance by using multiple disks would then require tens of gigabytes of RAM.

Reducing the RAM needed for indexing is desirable for several scenarios. Since the growth in disk capacities has been exceeding the growth of RAM capacity for some time, this trend will lead to systems where the disk cannot be fully indexed due to a lack of RAM. Dedicated RAM also effectively limits the degree of multiprogramming of the system, so as processors get faster relative to network speeds, one may wish to consolidate multiple functions on a single server. WAN accelerators, for example, cache network data [5, 29, 34], so having very large storage can reduce bandwidth consumption more than HTTP proxies alone. Similarly, even in HTTP proxies, RAM may be more useful as a hot object cache than as an index, as is the case in reverse proxies (server accelerators) and content distribution networks. One goal in designing HashCache is to determine how much index memory is really necessary.

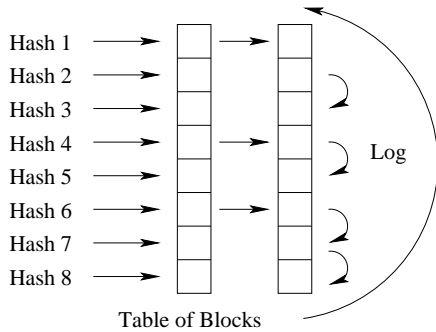


Figure 1: HashCache-Basic: objects with hash value i go to the i^{th} bin for the first block of a file. Later blocks are in the circular log.

3 Design

In this section, we present the design of HashCache and show how performance can be scaled with available memory. We begin by showing how to eliminate the in-memory index while still obtaining reasonable performance, and then we show how selective use of minimal indexing can improve performance. A summary of policies is shown in Table 3.

3.1 Removing the In-Memory Index

We start by removing the in-memory index entirely, and incrementally introducing minimal metadata to systematically improve performance. To remove the in-memory index, we have to address the two functions the in-memory index serves: indicating the existence of an object and specifying its location on disk. Using filesystem directories to store objects by hash has its own performance problems, so we seek an alternative solution – treating the disk as a simple hashtable.

HashCache-Basic, the simplest design option in the HashCache family, treats part of the disk as a fixed-size, non-chained hash table, with one object stored in each bin. This portion is called the Disk Table. It hashes the object name (a URL in the case of a Web cache) and then calculates the hash value modulo the number of bins to determine the location of the corresponding file on disk. To avoid false positives from hash collisions, each stored object contains metadata, including the original object name, which is compared with the requested object name to confirm an actual match. New objects for a bin are simply written over any previous object.

Since objects may be larger than the fixed-size bins in the Disk Table, we introduce a circular log that contains the remaining portion of large objects. The object metadata stored in each Disk Table bin also includes the location in the log, the object size, and the log generation number, and is illustrated in Figure 1.

The performance impact of these decisions is as follows: in comparison to high-performance caches,

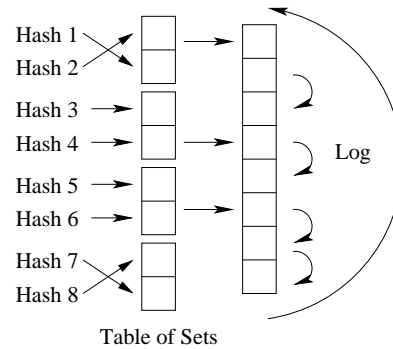


Figure 2: HashCache-Set: Objects with hash value i search through the $\frac{i}{N}^{th}$ set for the first block of a file. Later blocks are in the circular log. Some arrows are shown crossed to illustrate that objects that map on to a set can be placed anywhere in the set.

HashCache-Basic will have an increase in hash collisions (reducing cache hit rates), and will require a disk access on every request, even cache misses. Storing objects will require one seek per object (due to the hash randomizing the location), and possibly an additional write to the circular log.

3.2 Collision Control Mechanism

While in-memory indexes can use hash chaining to eliminate the problem of hash values mapped to the same bin, doing so for an on-disk index would require many random disk seeks to walk a hash bin, so we devise a simpler and more efficient approach while retaining most of the benefits.

In HashCache-Set, we expand the Disk Table to become an N -way set-associative hash table, where each bin can store N elements. Each element still contains metadata with the full object name, size, and location in the circular log of any remaining part of the object. Since these locations are contiguous on disk, and since short reads have much lower latency than seeks, reading all of the members of the set takes only marginally more time than reading just one element. This approach is shown in Figure 2, and reduces the impact of popular objects mapping to the same hash bin, while only slightly increasing the time to access an object.

While HashCache-Set eliminates problems stemming from collisions in the hash bins, it still has several problems: it requires disk access for cache misses, and lacks an efficient mechanism for cache replacement within the set. Implementing something like LRU within the set using the on-disk mechanism would require a potential disk write on every cache hit, reducing performance.

3.3 Avoiding Seeks for Cache Misses

Requiring a disk seek to determine a cache miss is a major issue for workloads with low cache hit rates, since an

Policy	Bits Per Object	RAM GB per Disk TB	Read Seeks	Write Seeks	Miss Seeks	Comments
Squid	576-832	9 - 13	~ 6	~ 6	0	Harvest descendant
Commercial	256-544	4 - 8.5	< 1	~ 0	0	custom filesystem
HC-Basic	0	0	1	1	1	high collision rate
HC-Set	0	0	1	1	1	adds N-way sets to reduce collisions
HC-SetMem	11	0.17	1	1	0	small in-mem hash eliminates miss seeks
HC-SetMemLRU	< 11	< 0.17	1	1	< 1	only some sets kept in memory
HC-Log	47	0.73	1	~ 0	0	writes to log, log position added to entry
HC-LogLRU	15-47	0.23 - 0.67	1 + ϵ	~ 0	0	log position for only some entries in set
HC-LogLRU + Prefetch	23-55	0.36 - 0.86	< 1	~ 0	0	reads related objects together
HC-Log + Prefetch	55	0.86	< 1	~ 0	0	reads related objects together

Table 3: Summary of HashCache policies, with Squid and commercial entries included for comparison. Main memory consumption values assume an average object size of 8KB. Squid memory data appears in <http://www.comfsm.fm/computing/squid/FAQ-8.html>

index-less cache would spend most of its disk time confirming cache misses. This behavior would add extra latency for the end-user, and provide no benefit. To address the problem of requiring seeks for cache misses, we introduce the first HashCache policy with any in-memory index, but employ several optimizations to keep the index much smaller than traditional approaches.

As a starting point, we consider storing in main memory an H-bit hash values for each cached object. These hash values can be stored in a two-dimensional array which corresponds to the Disk Table, with one row for each bin, and N columns corresponding to the N-way associativity. An LRU cache replacement policy would need forward and reverse pointers per object to maintain the LRU list, bringing the per-object RAM cost to $(H + 64)$ bits assuming 32-bit pointers. However, we can reduce this storage as follows.

First, we note that all the entries in an N-entry set share the same modulo hash value ($\%S$) where S is the number of sets in the Disk Table. We can drop the lowest $\log(S)$ bits from each hash value with no loss, reducing the hash storage to only $H - \log(S)$ bits per object.

Secondly, we note that cache replacement policies only need to be implemented within the N-entry set, so LRU can be implemented by simply ranking the entries from 0 to N-1, thereby using only $\log(N)$ bits per entry.

We can further choose to keep in-memory indexes for only some sets, not all sets, so we can restrict the number of in-memory entries based on request rate, rather than cache size. This approach keeps sets in an LRU fashion, and fetches the in-memory index for a set from disk on demand. By keeping only partial sets, we need to also keep a bin number with each set, LRU pointers per set, and a hash table to find a given set in memory.

Deciding when to use a complete two-dimensional array versus partial sets with bin numbers and LRU pointers depends on the size of the hash value and the set associativity. Assuming 8-way associativity and the 8 most

significant hash bits per object, the break-even point is around 50% – once more than half the sets will be stored in memory, it is cheaper to remove the LRU pointers and bin number, and just keep all of the sets. A discussion of how to select values for these parameters is provided in Section 4.

If the full array is kept in memory, we call it HashCache-SetMem, and if only a subset are kept in memory, we call it HashCache-SetMemLRU. With a low hash collision rate, HashCache-SetMem can determine most cache misses without accessing disk, whereas HashCache-SetMemLRU, with its tunable memory consumption, will need disk accesses for some fraction of the misses. However, once a set is in memory, performing intra-set cache replacement decisions requires no disk access for policy maintenance. Writing objects to disk will still require disk access.

3.4 Optimizing Cache Writes

With the previous optimizations, cache hits require one seek for small files, and cache misses require no seeks (excluding false positives from hash collisions) if the associated set’s metadata is in memory. Cache writes still require seeks, since object locations are dictated by their hash values, leaving HashCache at a performance disadvantage to high-performance caches that can write all content to a circular log. This performance problem is not an issue for caches with low request rates, but will become a problem for higher request rate workloads.

To address this problem, we introduce a new policy, HashCache-Log, that eliminates the Disk Table and treats the disk as a log, similar to the high-performance caches. For some or all objects, we store an additional offset (32 or 64 bits) specifying the location on disk. We retain the N-way set associativity and per-set LRU replacement because they eliminate disk seeks for cache misses with compact implementation. While this approach significantly increases memory consumption, it

can also yield a large performance advantage, so this tradeoff is useful in many situations. However, even when adding the log location, the in-memory index is still much smaller than traditional caches. For example, for 8-way set associativity, per-set LRU requires 3 bits per entry, and 8 bits per entry can minimize hash collisions within the set. Adding a 32-bit log position increases the per-entry size from 11 bits to 43 bits, but virtually eliminates the impact of write traffic, since all writes can now be accumulated and written in one disk seek. Additionally, we need a few bits (assume 4) to record the log generation number, driving the total to 47 bits. Even at 47 bits per entry, HashCache-Log still uses indexes that are a factor of 6-12 times smaller than current high-performance proxies.

We can reduce this overhead even further if we exploit Web object popularity, where half of the objects are rarely, if ever, re-referenced [8]. In this case, we can drop half of the log positions from the in-memory index, and just store them on disk, reducing the average per-entry size to only 31 bits, for a small loss in performance. HashCache-LogLRU allows the number of log position entries per set to be configured, typically using $\frac{N}{2}$ log positions per N-object set. The remaining log offsets in the set are stored on the disk as a small contiguous file. Keeping this file and the in-memory index in sync requires a few writes reducing the performance by a small amount. The in-memory index size, in this case, is 9-20 times smaller than traditional high-performance systems.

3.5 Prefetching Cache Reads

With all of the previous optimizations, caching storage can require as little as 1 seek per object read for small objects, with no penalty for cache misses, and virtually no cost for cache writes that are batched together and written to the end of the circular log. However, even this performance can be further improved, by noting that prefetching multiple objects per read can amortize the read cost per object.

Correlated access can arise in situations like Web pages, where multiple small objects may be embedded in the HTML of a page, resulting in many objects being accessed together during a small time period. Grouping these objects together on disk would reduce disk seeks for reading and writing. The remaining blocks for these pages can all be coalesced together in the log and written together so that reading them can be faster, ideally with one seek.

The only change necessary to support this policy is to keep a content length (in blocks) for all of the related content written at the same time, so that it can be read together in one seek. When multiple related objects are read together, the system will perform reads at less than one seek per read on average. This approach can

Policy	Throughput
HC-Basic	$rr = \frac{t}{1 + \frac{1}{rel} + (1 - chr) \cdot cbr}$
HC-Set	$rr = \frac{t}{1 + \frac{1}{rel} + (1 - chr) \cdot cbr}$
HC-SetMem	$rr = \frac{t}{chr \cdot (1 + \frac{1}{rel}) + (1 - chr) \cdot cbr}$
HC-LogN	$rr = \frac{t}{2 \cdot chr + (1 - chr) \cdot cbr}$
HC-LogLRU	$rr = \frac{t \cdot rel}{2 \cdot chr + (1 - chr) \cdot cbr}$
HC-Log	$rr = \frac{t \cdot rel}{2 \cdot chr + (1 - chr) \cdot cbr}$
Commercial	$rr = \frac{t \cdot rel}{2 \cdot chr + (1 - chr) \cdot cbr}$

Table 4: Throughputs for techniques, rr = peak request rate, chr = cache hit rate, cbr = cacheability rate, rel = average number of related objects, t = peak disk seek rate – all calculations include read prefetching, so the results for Log and Grouped are the same. To exclude the effects of read prefetching, simply set rel to one.

be applied to many of the previously described HashCache policies, and only requires that the application using HashCache provide some information about which objects are related. Assuming prefetch lengths of no more than 256 blocks, this policy only requires 8 bits per index entry being read. In the case of HashCache-LogLRU, only the entries with in-memory log position information need the additional length information. Otherwise, this length can also be stored on disk. As a result, adding this prefetching to HashCache-LogLRU only increases the in-memory index size to 35 bits per object, assuming half the entries of each set contain a log position and prefetch length.

For the rest of this paper, we assume all the policies to have this optimization except HashCache-LogN which is the HashCache-Log policy without any prefetching.

3.6 Expected Throughput

To understand the throughput implications of the various HashCache schemes, we analyze their expected performance under various conditions using the parameters shown in Table 4.

The maximum request rate (rr) is a function of the disk seek rate, the hit rate, the miss rate, and the write rate. The write rate is required because not all objects that are fetched due to cache misses are cacheable. Table 4 presents throughputs for each system as a function of these parameters. The cache hit rate (chr) is simply a number between 0 and 1, as is the cacheability rate (cbr). Since the miss rate is $(1 - chr)$, the write rate can be represented as $(1 - chr) \cdot cbr$. The peak disk seek rate (t) is a measured quantity that is hardware-dependent, and the average number of related objects (rel) is always a positive number. Due to space constraints, we omit the derivations for these calculations. These throughputs are

conservative estimates because we do not take into account the in-memory hot object cache, where some portion of the main memory is used as a cache for frequently used objects, which can further improve throughput.

4 HashCache Implementation

We implement a common HashCache filesystem I/O layer so that we can easily use the same interface with different applications. We expose this interface via POSIX-like calls, such as `open()`, `read()`, `write()`, `close()`, `seek()`, etc., to operate on files being cached. Rather than operate directly on raw disk, HashCache uses a large file in the standard Linux ext2/ext3 filesystem, which does not require root privilege. Creating this zero-filled large file on a fresh ext2/ext3 filesystem typically creates a mostly contiguous on-disk layout. It creates large files on each physical disk and multiplexes them for performance. The HashCache filesystem is used by the HashCache Web proxy cache as well as other applications we are developing.

4.1 External Indexing Interface

HashCache provides a simple indexing interface to support other applications. Given a key as input, the interface returns a data structure containing the file descriptors for the Disk Table file and the contiguous log file (if required), the location of the requested content, and metadata such as the length of the contiguous blocks belonging to the item, etc. We implement the interface for each indexing policy we have described in the previous section. Using the data returned from the interface one can utilize the POSIX calls to handle data transfers to and from the disk. Calls to the interface can block if disk access is needed, but multiple calls can be in flight at the same time. The interface consists of roughly 600 lines of code, compared to 21000 lines for the HashCache Web Proxy.

4.2 HashCache Proxy

The HashCache Web Proxy is implemented as an event-driven main process with cooperating helper processes/threads handling all blocking operations, such as DNS lookups and disk I/Os, similar to the design of Flash [25]. When the main event loop receives a URL request from a client, it searches the in-memory hot-object cache to see if the requested content is already in memory. In case of a cache miss, it looks up the URL using one of the HashCache indexing policies. Disk I/O helper processes use the HashCache filesystem I/O interface to read the object blocks into memory or to write the fetched object to disk. To minimize inter-process communication (IPC) between the main process and the helpers, only beacons are exchanged on IPC channels and the actual data transfer is done via shared memory.

4.3 Flexible Memory Management

HTTP workloads will often have a small set of objects that are very popular, which can be cached in main memory to serve multiple requests, thus saving disk I/O. Generally, the larger the in-memory cache, the better the proxy's performance. HashCache proxies can be configured to use all the free memory on a system without unduly harming other applications. To achieve this goal, we implement the hot object cache via anonymous `mmap()` calls so that the operating system can evict pages as memory pressure dictates. Before the HashCache proxy uses the hot object cache, it checks the memory residency of the page via the `mincore()` system call, and simply treats any missing page as a miss in the hot object cache. The hot object cache is managed as an LRU list and unwanted objects or pages no longer in main memory can be unmapped. This approach allows the HashCache proxy to use the entire main memory when no other applications need it, and to seamlessly reduce its memory consumption when there is memory pressure in the system.

In order to maximize the disk writing throughput, the HashCache proxy buffers recently-downloaded objects so that many objects can be written in one batch (often to a circular log). These dirty objects can be served from memory while waiting to be written to disk. This dirty object cache reduces redundant downloads during flash crowds because many popular HTTP objects are usually requested by multiple clients.

HashCache also provides for grouping related objects to disk so that they can be read together later, providing the benefits of prefetching. The HashCache proxy uses this feature to amortize disk seeks over multiple objects, thereby obtaining higher read performance. One commercial system parses HTML to explicitly find embedded objects [7], but we use a simpler approach – simply grouping downloads by the same client that occur within a small time window and that have the same HTTP Referrer field. We have found that this approach works well in practice, with much less implementation complexity.

4.4 Parameter Selection

For the implementation, we choose some design parameters such as the block size, the set size, and the hash size. Choosing the block size is a tradeoff between space usage and the number of seeks necessary to read small objects. In Table 5, we show an analysis of object sizes from a live, widely-used Web cache called CoDeeN [41]. We see that nearly 75% of objects are less than 8KB, while 87.2% are less than 16KB. Choosing an 8KB block would yield better disk usage, but would require multiple seeks for 25% of all objects. Choosing the larger block size wastes some space, but may increase performance.

Since the choice of block size influences the set size,

Size (KB)	% of objects < size
8	74.8
16	87.2
32	93.8
64	97.1
128	98.8
256	99.5

Table 5: CDF of Web object sizes

we make the decisions based on the performance of current disks. Table 6 shows the average number of seeks per second of three recent SATA disks (18, 60 and 150 GB each). We notice the sharp degradation beyond 64KB, so we use that as the set size. Since 64KB can hold 4 blocks of 16KB each or 8 blocks of 8KB each, we opt for an 8KB block size to achieve 8-way set associativity. With 8 objects per set, we choose to keep 8 bits of hash value per object for the in-memory indexes, to reduce the chance of collisions. This kind of an analysis can be automatically performed during initial system configuration, and are the only parameters needed once the specific HashCache policy is chosen.

5 Performance Evaluation

In this section, we present experimental results that compare the performance of different indexing mechanisms presented in Section 3. Furthermore, we present a comparison between the HashCache Web Proxy Cache, Squid, and a high-performance commercial proxy called Tiger, using various configurations. Tiger implements the best practices outlined in Section 2 and is currently used in commercial service [6]. We also present the impact of the optimizations that we included in the HashCache Web Proxy Cache. For fair comparison, we use the same basic code base for all the HashCache variants, with differences only in the indexing mechanisms.

5.1 Workload

To evaluate these systems, we use the Web Polygraph [37] benchmarking tool, the *de facto* industry standard for testing the performance of HTTP intermediaries such as content filters and caching proxies. We use the Polymix [38] environment models, which models many key Web traffic characteristics, including: multiple content types, diurnal load spikes, URLs with transient popularity, a global URL set, flash crowd behavior, an unlimited number of objects, DNS names in URLs, object life-cycles (expiration and last-modification times), persistent connections, network packet loss, reply size variations, object popularity (recurrence), request rates and inter-arrival times, embedded objects and browser behavior, and cache validation (If-Modified-Since requests and reloads).

Read Size (KB)	Seeks/sec	Latency/seek (ms)
1	78	12.5
4	76	12.9
8	76	13.1
16	74	13.3
32	72	13.7
64	70	14.1
128	53	19.2

Table 6: Disk performance statistics

We use the latest standard workload, Polymix-4 [38], which was used at the Fourth Cache-off event [39] to benchmark many proxies. The Polygraph test harness uses several machines for emulating HTTP clients and others to act as Web servers. This workload offers a cache hit ratio (CHR) of 60% and a byte hit ratio (BHR) of 40% meaning that at most 60% of the objects are cache hits while 40% of bytes are cache hits. The average download latency is 2.5 seconds (including RTT). A large number of objects are smaller than 8.5 KB. HTML pages contain 10 to 20 embedded (related) objects, with an average size of 5 to 10 KB. A small number (0.1 %) of large downloads (300 KB or more) have higher cache hit rates. These numbers are very similar to the characteristics of traffic in developing regions [8].

We test three environments, reflecting the kinds of caches we expect to deploy. These are the low-end systems that reflect the proxy powered by a laptop or similar system, large-disk systems where a larger school can purchase external storage to pre-load content, and high-performance systems for ISPs and network backbones.

5.2 Low-End System Experiments

Our first test server for the proxy is designed to mimic a low-memory laptop, such as the OLPC XO Laptop, or a shared low-powered machine like an OLPC XS server. Its configuration includes a 1.4 GHz CPU with 512 KB of L2 cache, 256 MB RAM, two 60GB 7200 RPM SATA drives, and the Fedora 8 Linux OS. This machine is far from the standard commercial Web cache appliance, and is likely to be a candidate machine for the developing world [23].

Our tests for this machine configuration run at 40-275 requests per second, per disk, using either one or two disks. Figure 3 shows the results for single disk performance of the Web proxy using HashCache-Basic (HC-B), HashCache-Set (HC-S), HashCache-SetMem (HC-SM), HashCache-Log without object prefetching (HC-LN), HashCache-Log with object prefetching (HC-L), Tiger and Squid. The HashCache tests use 60 GB caches. However, Tiger and Squid were unable to index this amount of storage and still run acceptably, so were limited to using 18 GB caches. This smaller cache is still sufficient to hold the working set of the test, so Tiger and

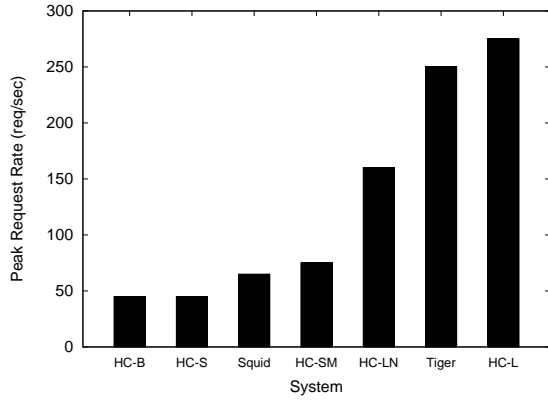


Figure 3: Peak Request Rates for Different policies for low end SATA disk.

policy	SATA 7200	SCSI 10000	SCSI 15000
HC-Basic	40	50	85
HC-Set	40	50	85
HC-SetMem	66	85	140
HC-LogN	132	170	280
HC-LogLRU	264	340	560
HC-Log	264	340	560
Commercial	264	340	560

Table 7: Expected throughputs (reqs/sec) for policies for different disk speeds— all calculations include read prefetching

Squid do not suffer in performance as a result. Table 7 gives the analytical lowerbounds for performance of each of these policies for this workload and the disk performance. The tests for HashCache-Basic and HashCache-Set achieve only 45 reqs/sec. The tests for HashCache-SetMem achieve 75 reqs/sec. Squid scales better than HashCache-Basic and HashCache-Set and achieves 60 reqs/sec. HashCache-Log (with prefetch), in comparison, achieves 275 reqs/sec. The Tiger proxy, with its optimized indexing mechanism, achieves 250 reqs/sec. This is less than HashCache-Log because Tiger’s larger index size reduces the amount of hot object cache available, reducing its prefetching effectiveness.

Figure 4 shows the results from tests conducted on HashCache-SetMem and two configurations of HashCache-SetMemLRU using 2 disks. The performance of the HashCache-SetMem system scales to 160 reqs/sec, which is slightly more than double its performance with a single disk. The reason for this difference is that the second disk does not have the overhead of handling all access logging for the entire system. The two other graphs in the figure, labeled HC-SML30 and HC-SML40, are the 2 versions of HashCache-SetMemLRU where only 30% and 40% of all the set headers are cached in main memory. As mentioned earlier, the

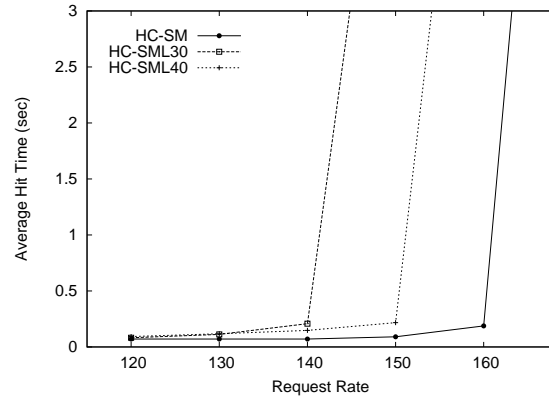


Figure 4: Peak Request Rates for Different SetMemLRU policies on low end SATA disks.

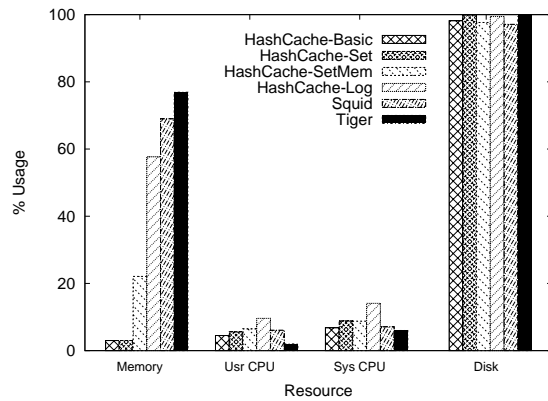


Figure 5: Resource Usage for Different Systems

hash table and the LRU list overhead of HashCache-SetMemLRU is such that when 50% of set headers are cached, it takes about the same amount of memory when using HashCache-SetMem. These experiments serve to show that HashCache-SetMemLRU can provide further savings when working set sizes are small and one does not need all the set headers in main memory at all times to perform reasonably well.

These experiments also demonstrate HashCache’s small systems footprint. Those measurements are shown in Figure 5 for the single-disk experiment. In all cases, the disk is the ultimate performance bottleneck, with nearly 100% utilization. The user and system CPU remain relatively low, with the higher system CPU levels tied to configurations with higher request rates. The most surprising metric, however, is Squid’s high memory usage rate. Given that its storage size was only one-third that used by HashCache, it still exceeds HashCache’s memory usage in HashCache’s highest-performance configuration. In comparison, the lowest-performance HashCache configurations, which have performance comparable to Squid, barely register in terms of memory usage.

	Request Rate per sec	Throughput Mb/s	Hit Time msec	All Time msec	Miss Time msec	CHR %	BHR %
HashCache-Log	2200	116.98	77	1147	2508	56.91	41.06
Tiger	2300	121.40	98	1150	2512	56.49	41.40
Squid	400	21.38	63	1109	2509	57.25	41.22

Table 8: Performance on a high end system

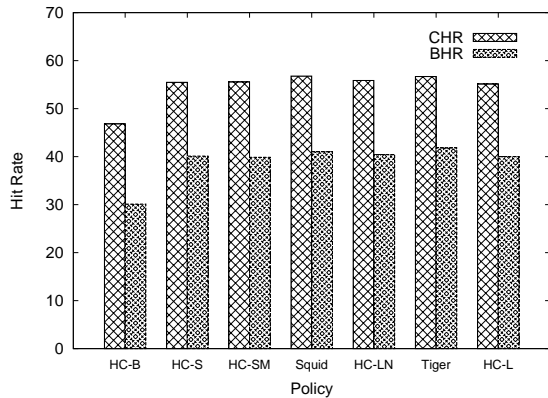


Figure 6: Low End Systems Hit Ratios

Figure 6 shows the cache hit ratio (by object) and the byte hit ratios (bandwidth savings) for the HashCache policies at their peak request rate. Almost all configurations achieve the maximum offered hit ratios, with the exception of HashCache-Basic, which suffers from hash collision effects.

While the different policies offer different tradeoffs, one might observe that the performance jump between HashCache-SetMem and HashCache-Log is substantial. To bridge this gap one can use multiple small disks instead of one large disk to increase performance while still using the same amount of main memory. These experiments further demonstrate that for low-end machines, HashCache can not only utilize more disk storage than commercial cache designs, but can also achieve comparable performance while using less memory. The larger storage size should translate into greater network savings, and the low resource footprint ensures that the proxy machine need not be dedicated to just a single task. The HashCache-SetMem configuration can be used when one wants to index larger disks on a low-end machine with a relatively low traffic demand. The lowest-footprint configurations, which use no main-memory indexing, HashCache-Basic and HashCache-Set, would even be appropriate for caching in wireless routers or other embedded devices.

5.3 High-End System Experiments

For our high-end system experiments, we choose hardware that would be more appropriate in a datacenter. The processor is a dual-core 2GHz Xeon, with 2MB of L2 cache. The server has 3.5GB of main memory, and

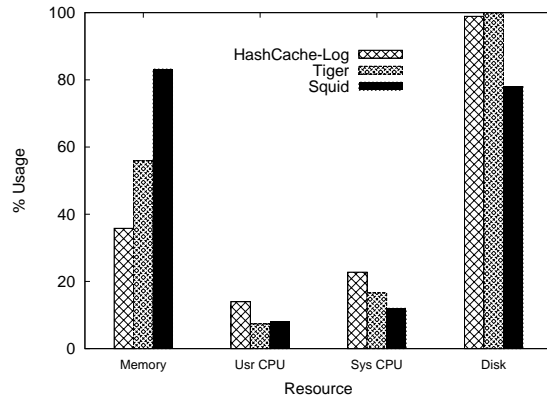


Figure 7: High End System Performance Statistics

five 10K RPM Ultra2 SCSI disks, of 18GB each. These disks perform 90 to 95 random seeks/sec. Using our analytical models, we expect a performance of at least 320 reqs/sec/disk with HashCache-Log. On this machine we run HashCache-Log, Tiger and Squid. From the HashCache configurations, we chose only HashCache-Log because the ample main memory of this machine would dictate that it can be used for better performance rather than maximum cache size.

Figure 7 shows the resource utilization of the three systems at their peak request rates. HashCache-Log consumes just enough memory for hot object caching, write buffers and also the index, still leaving about 65% of the memory unused. At the maximum request rate, the workload becomes completely disk bound. Since the working set size is substantially larger than the main memory size, expanding the hot object cache size produces diminishing returns. Squid fails to reach 100% disk throughput simultaneously on all disks. Dynamic load imbalance among its disks causes one disk to be the system bottleneck, even though the other four disks are underutilized. The load imbalance prevents it from achieving higher request rates or higher average disk utilization.

The performance results from this test are shown in Table 8, and they confirm the expectations from the analytical models. HashCache-Log and Tiger perform comparably well at 2200-2300 reqs/sec, while Squid reaches only 400 reqs/sec. Even at these rates, HashCache-Log is purely disk-bound, while the CPU and memory consumption has ample room for growth. The per-disk performance of HashCache-Log of 440 reqs/sec/disk is in

1TB Configuration	Request Rate per sec	Throughput Mb/s	Hit Time msec	All Time msec	Miss Time msec	CHR %	BHR %
HashCache-SetMem	75	3.96	27	1142	2508	57.12	40.11
HashCache-Log	300	16.02	48	1139	2507	57.88	40.21
HashCache-LogLRU	300	16.07	68	1158	2510	57.15	40.08
2TB Configuration	Request Rate per sec	Throughput Mb/s	Hit Time msec	All Time msec	Miss Time msec	CHR %	BHR %
HashCache-SetMem	150	7.98	32	1149	2511	57.89	40.89
HashCache-Log	600	32.46	56	1163	2504	57.01	40.07
HashCache-LogLRU	600	31.78	82	1171	2507	57.67	40.82

Table 9: Performance on large disks

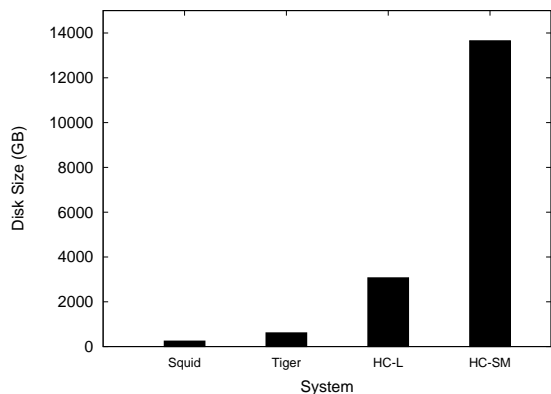


Figure 8: Sizes of disks that can be indexed by 2GB memory

line with the best commercial showings – the highest-performing system at the Fourth Cacheoff achieved less than an average of 340 reqs/sec/disk on 10K RPM SCSI disks. The absolute best throughput that we find from the Fourth Cacheoff results is 625 reqs/sec/disk on two 15K RPM SCSI disks, and on the same speed disks HashCache-Log and Tiger both achieve 700 reqs/sec/disk, confirming the comparable performance.

These tests demonstrate that the same HashCache code base can provide good performance on low-memory machines while matching or exceeding the performance of high-end systems designed for cache appliances. Furthermore, this performance comes with a significant savings in memory, allowing room for larger storage or higher performance.

5.4 Large Disk Experiments

Our final set of experiments involves using HashCache configurations with large external storage systems. For this test, we use two 1 TB external hard drives attached to the server via USB. These drives perform 67-70 random seeks per second. Using our analytical models, we would expect a performance of 250 reqs/sec with HashCache-Log. In other respects, the server is configured comparably to our low-end machine experiment, but the memory is increased from 256MB to 2GB to accommodate some

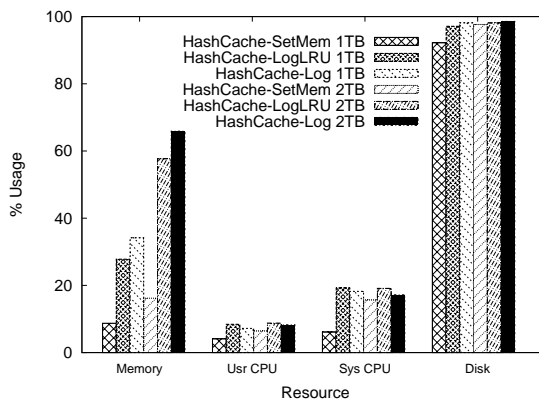


Figure 9: Large Disk System Performance Statistics

of the configurations that have larger index requirements, representative of low-end servers being deployed [24].

We compare the performance of HashCache-SetMem, HashCache-Log and HashCache-LogLRU with one or two external drives. Since the offered cache hit rate for the workload is 60%, we cache 6 out of the 8 log offsets in main memory for HashCache-LogLRU. For these experiments, the Disk Table is stored on a disk separate from the ones keeping the circular log. Also, since filling the 1TB hard drives at 300 reqs/second would take excessively long, we randomly place 50GB of data across each drive to simulate seek-limited behavior.

Unfortunately, even with 2GB of main memory, Tiger and Squid are unable to index these drives, so we were unable to test them in any meaningful way. Figure 8 shows the size of the largest disk that each of the systems can index with 2 GB of memory. In the figure, HC-SM and HC-L are HashCache-SetMem and HashCache-Log, respectively. The other HashCache configurations, Basic and Set have no practical limit on the amount of externally-attached storage.

The Polygraph results for these configurations are shown in Table 9, and the resource usage details are in Figure 9. With 2TB of external storage, both HashCache-Log and HashCache-LogLRU are able to perform 600 reqs/sec. In this configuration, HashCache-Log uses

slightly more than 60% of the system's memory, while HashCache-LogLRU uses slightly less. The hit time for HashCache-LogLRU is a little higher than HashCache-Log because in some cases it requires 2 seeks (one for the position, and one for the content) in order to perform a read. The slightly higher cache hit rates exhibited on this test versus the high-end systems test are due the Polygraph environment – without filling the cache, it has a smaller set of objects to reference, yielding a higher offered hit ratio.

The 1TB test achieves half the performance of the 2TB test, but does so with correspondingly less memory utilization. The HashCache-SetMem configuration actually uses less than 10% of the 2GB overall in this scenario, suggesting that it could have run with our original server configuration of only 256MB.

While the performance results are reassuring, these experiments prove that HashCache can index disks that are much larger than conventional policies could handle. At the same time, HashCache performance meets or exceeds what other caches would produce on much smaller disks. This scenario is particularly important for the developing world, because one can use these inexpensive high-capacity drives to host large amounts of content, such as a Wikipedia mirror, WAN accelerator chunks, HTTP cache, and any other content that can be preloaded or shipped on DVDs later.

6 Related Work

Web caching in its various forms has been studied extensively in the research and commercial communities. As mentioned earlier, the Harvest cache [4] and CERN caches [17] were the early approaches. The Harvest design persisted, especially with its transformation into the widely-used Squid Web proxy [35]. Much research has been performed on Squid, typically aimed at reorganizing the filesystem layout to improve performance [16, 18], better caching algorithms [14], or better use of peer caches [11]. Given the goals of HashCache, efficiently operating with very little memory and large storage, we have avoided more complexity in cache replacement policies, since they typically use more memory to make the decisions. In the case of working sets that dramatically exceed physical memory, cache policies are also likely to have little real impact. Disk cache replacement policies also become less effective when storage sizes grow very large. We have also avoided Bloom-filter approaches [2] that would require periodic rebuilds, since scanning terabyte-sized disks can sap disk performance for long periods. Likewise, approaches that require examining multiple disjoint locations [19, 32] are also not appropriate for this environment, since any small gain in reducing conflict misses would be offset by large losses in checking multiple locations on each cache miss.

Some information has been published about commercial caches and workloads in the past, including the design considerations for high-speed environments [3], proxy cache performance in mixed environments [9], and workload studies of enterprise user populations [12]. While these approaches have clearly been successful in the developed world, many of the design techniques have not typically transitioned to the more price-sensitive portions of the design space. We believe that HashCache demonstrates that addressing problems specific to the developing world can also open interesting research opportunities that may apply to systems that are not as price-sensitive or resource-constrained.

In terms of performance optimizations, two previous systems have used some form of prefetching, including one commercial system [7], and one research project [33]. Based on published metrics, HashCache performs comparably to the commercial system, despite using a much similar approach to grouping objects, and despite using a standard filesystem for storage instead of raw disk access. Little scalability information is presented on the research system, since it was tested only using Apache mod_proxy at 8 requests per second. Otherwise, very little information is publically available regarding how high-performance caches typically operate from the extremely competitive commercial period for proxy caches, centered around the year 2000. In that year, the Third Cache-Off [40] had a record number of vendors participate, representing a variety of different caching approaches. In terms of performance, HashCache-Log compares favorably to all of them, even when normalized for hardware.

Web caches also get used in two other contexts: server accelerators and content distribution networks (CDNs) [1, 10, 26, 41]. Server accelerators, also known as reverse proxies, typically reside in front of a Web server and offload cacheable content, allowing the Web server to focus on dynamically-generated content. CDNs geographically distribute the caches reducing latency to the client and bandwidth consumption at the server. In these cases, the proxy typically has a very high hit rate, and is often configured to serve as much content from memory as possible. We believe that HashCache is also well-suited for this approach, because in the Set-MemLRU configuration, only the index entries for popular content need to be kept in memory. By freeing the main memory from storing the entire index, the extra memory can be used to expand the size of the hot object cache.

Finally, in terms of context in developing world projects, HashCache is simply one piece of the infrastructure that can help these environments. Advances in wireless network technologies, such as WiMax [42] or rural WiFi [27, 36] will help make networking available

to larger numbers of people, and as demand grows, we believe that the opportunities for caching increase. Given the low resource usage of HashCache and its suitability for operation on shared hardware, we believe it is well-suited to take advantage of networking advancements in these communities.

7 Deployments

HashCache is currently deployed at two different locations in Africa, at the Obafemi Awolowo University (OAU) in Nigeria and at the Kokrobitey Institute (KI) in Ghana. At OAU, it runs on their university server which has a 100 GB hard drive, 2 GB memory and a dual core Xeon processor. For Internet connection, they pay \$5,000 per month for a 2 Mbps satellite link to an ISP in Europe and the link has a high variance ICMP ping time from Princeton ranging 500 to 1200 ms. We installed HashCache-Log on the machine but were asked to limit resource usage for HashCache to 50 GB disk space and no more than 300 MB of physical memory. The server is running other services such as a E-mail service and a firewall for the department and it is also used for general computation for the students. Due to privacy issues we were not able to analyze the logs from this deployment but the administrator has described the system as useful and also noticed the significant memory and CPU usage reduction when compared to Squid.

At KI, HashCache runs on a wireless router for a small department on a 2 Mbps LAN. The Internet connection is through a 256 Kbps sub-marine link to Europe and the link has a ping latency ranging from 200 to 500 ms. The router has a 30 GB disk and 128 MB of main memory and we were asked to use 20 GB of disk space and as little memory as possible. This prompted us to use the HashCache-Set policy as there are only 25 to 40 people using the router every day. Logging is disabled on this machine as well since we were asked not to consume network bandwidth on transferring the logs.

In both these deployments we have used HashCache policies to improve the Web performance while consuming minimum amount of resource. Other solutions like Squid would not have been able to meet these resource constraints while providing any reasonable service. People at both places told us that the idea of a faster Internet to popular Web sites seemed like a distant dream until we discussed the complete capabilities of HashCache. We are currently working with OLPC to deploy HashCache at more locations with the OLPC XS servers.

8 Conclusion and Future Work

In this paper we have presented HashCache, a high-performance configurable cache storage for the developing regions. HashCache provides a range of configurations that scale from using no memory for indexing

to ones that require only one-tenth as much as current high-performance approaches. It provides this flexibility without sacrificing performance – its lowest-resource configuration has performance comparable to free software systems, while its high-end performance is comparable to the best commercial systems. These configurations allow memory consumption and performance to be tailored to application needs, and break the link between storage size and in-memory index size that has been commonly used in caching systems for the past decade. The benefits of HashCache’s low resource consumption allow it to share hardware with other applications, share the filesystem, and to scale to storage sizes well beyond what present approaches provide.

On top of the HashCache storage layer, we have built a Web caching proxy, the HashCache Proxy, which can run using any of the HashCache configurations. Using industry-standard benchmarks and a range of hardware configurations, we have shown that HashCache performs competitively with existing systems across a range of workloads. This approach provides an economy of scale in HashCache deployments, allowing it to be powered from laptops, low-resource desktops, and even high-resource servers. In all cases, HashCache either performs competitively or outperforms other systems suited to that class of hardware.

With its operation flexibility and a range of available performance options, HashCache is well suited to providing the infrastructure for caching applications in developing regions. Not only does it provide competitive performance with the stringent resource constraint, but also enables new opportunities that were not possible with existing approaches. We believe that HashCache can become the basis for a number of network caching services, and are actively working toward this goal.

9 Acknowledgements

We would like to thank Jim Gettys and John Watlington for their discussions about OLPC’s caching needs, and Marc Ficuzynski for arranging and coordinating our deployments in Africa. We also thank our shepherd, Michael Mitzenmacher as well as anonymous NSDI reviewers. This research was partially supported by NSF Awards CNS-0615237, CNS-0519829, and CNS-0520053. Anirudh Badam was partially supported by a Technology for Developing Regions Fellowship from Princeton University.

References

- [1] AKAMAI TECHNOLOGIES INC. <http://www.akamai.com/>.
- [2] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13 (1970), 422–426.

- [3] BREWER, E., GAUTHIER, P., AND MCEVOY, D. Long-term viability of large-scale caches. In *Proceedings of the 3rd International WWW Caching Workshop* (1998).
- [4] CHANKHUNTHOD, A., DANZIG, P. B., NEERDAELS, C., SCHWARTZ, M. F., AND WORRELL, K. J. A hierarchical internet object cache. In *Proceedings of the USENIX Annual Technical Conference* (1996).
- [5] CITRIX SYSTEMS. <http://www.citrix.com/>.
- [6] COBLITZ, INC. <http://www.coblitz.com/>.
- [7] COX, A. L., HU, Y. C., PAI, V. S., PAI, V. S., AND ZWAENEPOEL, W. Storage and retrieval system for WEB cache. U.S. Patent 7231494, 2000.
- [8] DU, B., DEMMER, M., AND BREWER, E. Analysis of WWW traffic in Cambodia and Ghana. In *Proceedings of the 15th International conference on World Wide Web (WWW)* (2006).
- [9] FELDMANN, A., CACERES, R., DOUGLIS, F., GLASS, G., AND RABINOVICH, M. Performance of web proxy caching in heterogeneous bandwidth environments. In *Proceedings of the 18th IEEE INFOCOM* (1999).
- [10] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIERES, D. Democratizing content publication with coral. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2004).
- [11] GADDE, S., CHASE, J., AND RABINOVICH, M. A taste of crispy Squid. In *Workshop on Internet Server Performance* (1998).
- [12] GRIBBLE, S., AND BREWER, E. A. System design issues for internet middleware services: Deductions from a large client trace. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)* (1997).
- [13] INTEL. Classmate PC, <http://www.classmatepc.com/>.
- [14] JIN, S., AND BESTAVROS, A. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)* (2000).
- [15] LUOTONEN, A., HENRYK, F., LEE, T. B. <http://info.cern.ch/hypertext/WWW/Daemon/Status.html>.
- [16] MALTZAHN, C., RICHARDSON, K., AND GRUNWALD, D. Reducing the disk I/O of Web proxy server caches. In *Proceedings of the USENIX Annual Technical Conference* (1999).
- [17] MALTZAHN, C., RICHARDSON, K. J., AND GRUNWALD, D. Performance issues of enterprise level web proxies. In *Proceedings of the ACM SIGMETRICS* (1997).
- [18] MARKATOS, E. P., PNEVMATIKATOS, D. N., FLOURIS, M. D., AND KATEVENIS, M. G. Web-conscious storage management for web proxies. *IEEE/ACM Transactions on Networking* 10, 6 (2002), 735–748.
- [19] MITZENMACHER, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.
- [20] MOGUL, J. C., CHAN, Y. M., AND KELLY, T. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2004).
- [21] O3B NETWORKS. <http://www.o3bnetworks.com/>.
- [22] OLPC. <http://www.laptop.org/>.
- [23] OLPC. http://wiki.laptop.org/go/Hardware_specification.
- [24] OLPC. http://wiki.laptop.org/go/XS_Recommended_Hardware.
- [25] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable Web server. In *Proceedings of the USENIX Annual Technical Conference* (1999).
- [26] PARK, K., AND PAI, V. S. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2006).
- [27] PATRA, R., NEDEVSCHI, S., SURANA, S., SHETH, A., SUBRAMANIAN, L., AND BREWER, E. WILDNet: Design and implementation of high performance wifi based long distance networks. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)* (2007).
- [28] RHEA, S., LIANG, K., AND BREWER, E. Value-based web caching. In *In Proceeding of the 13th International Conference on World Wide Web (WWW)* (2003).
- [29] RIVERBED TECHNOLOGY, INC. <http://www.riverbed.com/>.
- [30] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* 10, 1 (1992), 26–52.
- [31] RUSSELL, M., AND HOPKINS, T. CFTP: a caching FTP server. *Computer Networks and ISDN Systems* 30, 22–23 (1998), 2211–2222.
- [32] SEZNEC, A. A case for two-way skewed-associative caches. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 1993), ACM, pp. 169–178.
- [33] SHRIVER, E. A. M., GABBER, E., HUANG, L., AND STEIN, C. A. Storage management for web proxies. In *Proceedings of the USENIX Annual Technical Conference* (2001).
- [34] SILVER PEAK SYSTEMS, INC. <http://www.silver-peak.com/>.
- [35] SQUID. <http://www.squid-cache.org/>.
- [36] SUBRAMANIAN, L., SURANA, S., PATRA, R., NEDEVSCHI, S., HO, M., BREWER, E., AND SHETH, A. Rethinking wireless in the developing world. In *Proceedings of Hot Topics in Networks (HotNets-V)* (2006).
- [37] THE MEASUREMENT FACTORY. <http://www.web-polygraph.org/>.
- [38] THE MEASUREMENT FACTORY. <http://www.web-polygraph.org/docs/workloads/polymix-4/>.
- [39] THE MEASUREMENT FACTORY. <http://www.measurement-factory.com/results/public/cacheoff/N04/report.by-alpha.html>.
- [40] THE MEASUREMENT FACTORY. <http://www.measurement-factory.com/results/public/cacheoff/N03/report.by-alpha.html>.
- [41] WANG, L., PARK, K., PANG, R., PAI, V., AND PETERSON, L. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference* (2004).
- [42] WiMAX FORUM. <http://www.wimaxforum.org/home/>.

SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy

Anirudh Badam and Vivek S. Pai
Princeton University

Abstract

We introduce SSDAlloc, a hybrid main memory management system that allows developers to treat solid-state disk (SSD) as an extension of the RAM in a system. SSDAlloc moves the SSD upward in the memory hierarchy, usable as a larger, slower form of RAM instead of just a cache for the hard drive. Using SSDAlloc, applications can nearly transparently extend their memory footprints to hundreds of gigabytes and beyond without restructuring, well beyond the RAM capacities of most servers. Additionally, SSDAlloc can extract 90% of the SSD’s raw performance while increasing the lifetime of the SSD by up to 32 times. Other approaches either require intrusive application changes or deliver only 6–30% of the SSD’s raw performance.

1 Introduction

An increasing number of networked systems today rely on in-memory (DRAM) indexes, hashtables, caches and key-value storage systems for scaling the performance and reducing the pressure on their secondary storage devices. Unfortunately, the cost of DRAM increases dramatically beyond 64GB per server, jumping from a few thousand dollars to tens of thousands of dollars fairly quickly; power requirements scale similarly, restricting applications with large workloads from obtaining high in-memory hit-rates that are vital for high-performance.

Flash memory can be leveraged (by **augmenting** DRAM with flash backed memory) to scale the performance of such applications. Flash memory has a larger capacity, lower cost and lower power requirement when compared to DRAM and a great random read performance, which makes it well suited for building such applications. Solid State Disks (SSD) in the form of NAND flash have become increasingly popular due to pricing. 256GB SSDs are currently around \$700, and multiple SSDs can be placed in one server. As a result, high-end systems could easily augment their 64–128GB RAM with 1–2TB of SSD.

Flash is currently being used as program memory via two methods – by using flash as an operating system (OS) swap layer or by building a custom object store on top of flash. Swap layer, which works at a page granularity, reduces the performance and also undermines the

lifetime of flash for applications with many random accesses (typical of the applications mentioned). For every application object that is read/written (however small) an entire page of flash is read/dirtied leading to an unnecessary increase in the read bandwidth and the number of flash writes (which reduce the lifetime of flash memory). Applications are often modified to obtain high performance and good lifetime from flash memory by addressing these issues. Such modifications not only need a deep application knowledge but also require an expertise with flash memory, hindering a wide-scale adoption of flash. It is, therefore, necessary to expose flash via a swap like interface (via virtual memory) while being able to provide performance comparable to that of applications redesigned to be flash-aware.

In this paper, we present SSDAlloc, a **hybrid DRAM/flash memory manager** and a **runtime library** that allows applications to fully utilize the potential of flash (large capacity, low cost, fast random reads and non-volatility) in a transparent manner. SSDAlloc exposes flash memory via the familiar page-based virtual memory manager interface, but internally, it works at an object granularity for obtaining high performance and for maximizing the lifetime of flash memory. SSDAlloc’s memory manager is compatible with the standard C programming paradigms and it works entirely via the virtual memory system. Unlike object databases, applications do not have to declare their intention to use data, nor do they have to perform indirections through custom handles. All data maintains its virtual memory address for its lifetime and can be accessed using standard pointers. Pointer swizzling or other fix-ups are not required.

SSDAlloc’s memory allocator looks and feels much like the `malloc` memory manager. When `malloc` is directly replaced with SSDAlloc’s memory manager, flash is used as a fully log-structured page store. However, when SSDAlloc is provided with the additional information of the size of the application object being allocated, flash is managed as a log-structured object store. It utilizes the object size information to provide the applications with benefits that are otherwise unavailable via existing transparent programming techniques.

Using SSDAlloc, we have modified four systems built originally using `malloc`: memcached [4] (a key-value store), a Boost [1] based B+Tree index, a packet cache

Application	Original LOC	Edited LOC	Throughput Gain vs	
			SSD Swap Unmodified	SSD Swap Write Log
Memcached	11,193	21	5.5 - 17.4x	1.4 - 3.5x
B+Tree Index	477	15	4.3 - 12.7x	1.4 - 3.2x
Packet Cache	1,540	9	4.8 - 10.1x	1.3 - 2.3x
HashCache	20,096	36	5.3 - 17.1x	1.3 - 3.3x

Table 1: SSDAlloc requires changing only the memory allocation code, typically only tens of lines of code (LOC). Depending on the SSD used, throughput gains can be as high as 17 times greater than using the SSD as swap. Even if the swap is optimized for SSD usage, gains can be as high as 3.5x.

backend (for accelerating network links using packet level caching), and the HashCache [9] cache index. As shown in Table 1, all four systems show great benefits when using SSDAlloc with object size information –

- **4.1–17.4** times faster than when using the SSD as a swap space.
- **1.2–3.5** times faster than when using the SSD as a log-structured swap space.
- Only **9–36** lines of code are modified (`malloc` replaced by `SSDAlloc`).
- Up to **31.2** times less data written to the SSD for the same workload (SSDAlloc works at an object granularity).

The rest of this paper is organized as follows: We describe related work and the motivation in Section 2. The design is described in Section 3, and we discuss our implementation in Section 4. Section 5 provides the evaluation results, and we conclude in Section 6.

2 Motivation and Related Work

While alternative memory technologies have been championed for more than a decade [10, 25], their attractiveness has increased recently as the gap between the processor speed and the disk widened, and as their costs dropped. Our goal in this paper is to provide a transparent interface to using flash memory (unlike the application redesign strategy) while acting in a flash-aware manner to obtain better performance and lifetime from the flash device (unlike the operating system swap).

Existing transparent approaches to using flash memory [18, 20, 23] cannot fully exploit flash’s performance for two reasons – 1) Accesses to flash happen at a page granularity (4KB), leading to a full page read/write to flash for every access within that page. The write/erase behavior of flash memory often has different expectations on usage, leading to a poor performance. Full pages containing dirty objects have to be written to flash. This behavior leads to write escalation which is bad not only for performance but also for the durability of the flash device. 2) If the application objects are small compared to the page size, only a small fraction of RAM contains

useful objects because of caching at a page granularity. Integrating flash as a filesystem cache can increase performance, but the cost/benefit tradeoff of this approach has been questioned before [21].

FlashVM [23] is a system that proposes using flash as a dedicated swap device, that provides hints to the SSD for better garbage collection by batching writes, erases and discards. We propose using 16–32 times more flash than DRAM and in those settings, FlashVM style heuristic batching/aggregating of in-place writes might be of little use purely because of the high write randomness that our targeted applications have. A fully log-structured system would be needed for minimizing erases in such cases. We have built a fully log-structured swap that we use as a comparison point, along with native linux swap, against the SSDAlloc system that works at an object granularity.

Others have proposed redesigning applications to use flash-aware data structures to explicitly handle the asymmetric read/write behavior of flash. Redesigned applications range from databases (BTrees) [19, 24] and Web servers [17] to indexes [6, 8] and key-value stores [7]. Working set objects are cached in RAM more efficiently and the application aggregates objects when writing to flash. While the benefits of this approach can be significant, the costs involved and the extra development effort (requires expertise with the application and flash behavior) are high enough that it may deter most application developers from going this route.

Our goal in this paper is to provide the right set of interfaces (via memory allocators), so that both existing applications and new applications can be easily adapted to use flash. Our approach focuses on exposing flash only via a page based virtual memory interface while internally working at an object level. Similar approach was used in distributed object systems [12], which switched between pages and objects when convenient using custom object handlers. We want to avoid using any custom pointer/handler mechanisms to eliminate intrusive application changes.

Additionally, our approach can improve the cost/benefit ratio of flash-based approaches. If only a few lines of memory allocation code need to be modified to migrate an existing application to a flash-enabled one with performance comparable to that of flash-aware application redesign, this one-time development cost is low compared to the cost of high-density memory. For example, the cost of 1TB of high-density RAM adds roughly \$100K USD to the \$14K base price of the system (e.g., the Dell PowerEdge R910). In comparison, a high-end 320GB SSD sells for \$3200 USD, so roughly 4 servers with 5TB of flash memory cost the same as 1 server with 1 TB of RAM.

SSD Usage Technique	Write Logging	Read/Write < a page	Garbage Collects Dead pages/data	Avoids DRAM Pollution	Persistent Data	High Performance	Programming Ease
SSD Swap							✓
SSD Swap (Write Logged)	✓						✓
SSD mmap					✓		✓
Application Rewrite	✓	✓	✓	✓	✓	✓	✓
SSDAlloc	✓	✓	✓	✓	✓	✓	✓

Table 2: While using SSDs via swap/mmap is simple, they achieve only a fraction of the SSD’s performance. Rewriting applications can achieve greater performance but at a high developer cost. SSDAlloc provides simplicity while providing high performance.

SSD Make	reads / sec		writes / sec	
	4KB	0.5KB	4KB	0.5KB
RiDATA (32GB)	3,200	3,700	500	675
Kingston (64GB)	3,300	4,200	1,800	2,000
Intel X25-E (32GB)	26,000	44,000	2,200	2,700
Intel X25-V (40GB)	27,000	46,000	2,400	2,600
Intel X25-M G2 (80GB)	29,000	49,000	2,300	2,500

Table 3: SSDAlloc can take full advantage of object-sized accesses to the SSD, which can often provide significant performance gains over page-sized operations.

3 SSDAlloc’s Design

In this section we describe the design of SSDAlloc. We first start with describing the networked systems’ requirements from a hybrid DRAM/SSD setting for high-performance and ease of programming. Our high level goals for integrating SSDs into these applications are:

- To present a simple interface such that the applications can be run mostly unmodified – Applications should use the same programming style and interfaces as before (via virtual memory managers), which means that objects, once allocated, always appear to the application at the same locations in the virtual memory.
- To utilize the DRAM in the system as efficiently as possible – Since most of the applications that we focus on allocate large number of objects and operate over them with little locality of reference, the system should be no worse at using DRAM than a custom DRAM based object cache that efficiently packs as many hot objects in DRAM as possible.
- To maximize the SSD’s utility – Since the SSD’s read performance and especially the write performance suffer with the amount of data transferred, the system should minimize data transfers and (most importantly) avoid random writes.

SSDAlloc employs many clever design decisions and policies to meet our high level goals. In Sections 3.1 and 3.4, we describe our page-based virtual memory system using a modified heap manager in combination with a user-space on-demand page materialization runtime that appears to be a normal virtual memory

system to the application. In reality, the virtual memory pages are materialized in an on-demand fashion from the SSD by intercepting page faults. To make this interception as precise as possible, our allocator aligns the application level objects to always start at page boundaries. Such a fine grained interception allows our system to act at an application object granularity and thereby increases the efficiency of reads, writes and garbage collection on the SSD. It also helps in the design of a system that can easily serialize the application’s objects to the persistent storage for a subsequent usage.

In Section 3.2, we describe how we use the DRAM efficiently. Since most of the application’s objects are smaller than a page, it makes no sense to use all of the DRAM as a page cache. Instead, most of DRAM is filled with an object cache, which packs multiple useful objects per page, and one which is not directly accessible to the application. When the application needs a page, it is dynamically materialized, either from the object cache or from the SSD.

In Sections 3.3 and 3.5 we describe how we manage the SSD as an efficient log-structured object store. In order to reduce the amount of data read/written to the SSD, the system uses the object size information, given to the memory allocator by the application, to transfer only the objects, and not whole pages containing them. Since the objects can be of arbitrary sizes, packing them together and writing them in a log not only reduces the write volume, but also increase the SSD’s lifetime.

Table 2 presents an overview of various techniques by which SSDs are used as program memory today and provides a comparison to SSDAlloc by enumerating the high-level goals that each technique satisfies. We now describe our design in detail starting with our virtual address allocation policies.

3.1 SSDAlloc’s Virtual Memory Structure

SSDAlloc ideally wants to non-intrusively observe what objects the application reads and writes. The virtual memory (VM) system provides an easy way to detect what pages have been read or written, but there is no easy way to detect at a finer granularity. Performing copy-on-write and comparing the copy with the original can be used for detecting changes, but no easy mechanism de-

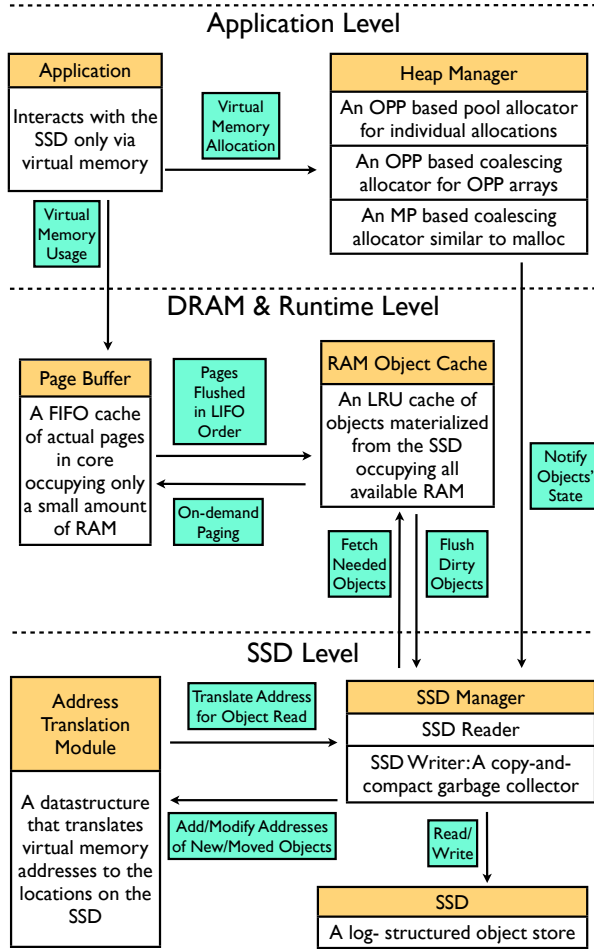


Figure 1: SSDAlloc uses most of RAM as an object-level cache, and materializes/dematerializes pages as needed to satisfy the application’s page usage. This approach improves RAM utilization, even though many objects will be spread across a greater range of virtual address space.

termines what parts of a page were read. Instead, SSDAlloc uses the observation that virtual address space is relatively inexpensive compared to actual DRAM, and reorganizes the behavior of memory allocation to use the VM system to observe object behavior. Servers typically expose 48 bit address spaces (256TB) while supporting less than 1TB of physical RAM, so virtual addresses are at least 256x more plentiful.

We propose the Object Per Page (OPP) model, using which, if an application requests memory for an object, the object is placed on its own page of virtual memory, yielding a single page for small objects, or more (contiguous) when the object exceeds the page size. The object is always placed at the start of the page and the rest of the page is not utilized for memory allocation. In reality, however, we employ various optimizations (de-

scribed in Section 3.2) to eliminate the physical memory wastage that can occur because of such a lavish virtual memory usage. An OPP memory manager can be implemented just by maintaining a pool of pages (details of the actual memory manager used are given in Section 3.4). OPP is suitable for individual object allocations, typical of the applications we focus on. OPP objects are stored on the SSD in a log-structured manner (details are explained in Section 3.5). Additionally, using virtual memory based page-usage information, we can accurately determine which objects are being read and written (since there is only one object per page). However, it is not straightforward to use arrays of objects in this manner. In an OPP array, each object is separated by the page’s size as opposed to the object’s size. While it is possible to allocate OPP arrays in such a manner, it would require some code modifications to be able to use arrays in which objects separated by page boundaries as opposed being separated by object boundaries. We describe later in Section 3.4 how an OPP based coalescing allocator can be used to allocate OPP based arrays.

3.1.1 Contiguous Array Allocations

In the C programming language, array allocations via `malloc/calloc` expect array elements to be contiguous. We present an option called Memory Pages (MP) which can do this. In MP, when the application asks for a certain amount of memory, SSDAlloc returns a pointer to a region of virtual address space with the size requested. We use a `ptmalloc` [5] style coalescing memory manager (further explained in Section 3.4) built on top of bulk allocated virtual memory pages (via `brk`) to obtain a system which can allocate C style arrays. Internally, however, the pages in this space are treated like page sized OPP objects. For the rest of the paper, we treat MP pages as page sized OPP objects.

While the design of OPP efficiently leverages the virtual memory system’s page level usage information to determine application object behavior, it could lead to DRAM space wastage because the rest of the page beyond the object would not be used. To eliminate this wastage, we organize the physical memory such that only a small portion of DRAM contains actual materializations of OPP pages (Page Buffer) while the rest of the available DRAM is used as a compact hot object cache.

3.2 SSDAlloc’s Physical Memory Structure

The SSDAlloc runtime system eases application transparency by allowing objects to maintain the same virtual address over their lifetimes, while their physical location may be in a temporarily-materialized physical page mapped to its virtual memory page in the Page Buffer, the RAM Object Cache, or the SSD. Not only does the runtime materialize physical pages as needed, but it also

reclaims them when their usage drops. We first describe how objects are cached compactly in DRAM.

RAM Object Cache – Objects are cached in *RAM object cache* in a compact manner. RAM object cache occupies available portion of DRAM while only a small part of DRAM is use for pages that are currently in use (shown in Figure 1). This decision provides several benefits – 1) Objects cached in RAM can be accessed much faster than the SSD, 2) By performing usage-based caching of objects instead of pages, the relatively small RAM can cache more useful objects when using OPP, and 3) Given the density trends of SSD and RAM, object caching is likely to continue being a useful optimization going forward.

RAM object cache is maintained in LRU fashion. It indexes objects using their virtual memory page address as the key. An OPP object in RAM object cache is indexed by its OPP page address, while an MP page (a 4KB OPP object) is indexed with its MP page address. In our implementation, we used a hashtable with the page address as the key for this purpose. Clean objects being evicted from the RAM object cache are deallocated while dirty objects being evicted are enqueued to the SSD writer mechanism (shown in Figure 1).

Page Buffer – Temporarily materialized pages (in physical memory) are collectively known as the Page Buffer. These pages are materialized in an on-demand fashion (described below). Page Buffer size is application configurable, but in most of the applications we tested, we found that a Page Buffer of size less than 25MB was sufficient to bring down the rate of page materializations per second to the throughput of the application. However, regardless of the size of the Page Buffer, physical memory wastage from using OPP has to be minimized. To minimize this wastage we make the rest of the active OPP physical page (portion beyond the object) a part of the RAM object cache. RAM object cache is implemented such that the shards of pages that materialize into physical memory are used for caching objects.

SSDAlloc’s Paging – For a simple user space implementation we implement the Page Buffer via memory protection. All virtual memory allocated using SSDAlloc is protected (via `mprotect`). A page usage is detected when the protection mechanism triggers a fault. The required page is then unprotected (only read or write access is given depending on the type of fault to be able to detect writes separately) and its data is then populated in the seg-fault handler – an OPP page is populated by fetching the object from RAM object cache or the SSD and placing it at the front of the page. An MP page is populated with a copy of the page (a page sized object) from RAM object cache or the SSD.

Pages dematerialized from Page Buffer are converted to objects. Those objects are pushed into the RAM object

cache, the page is then `madvised` to be not needed and finally, the page is reprotected (via `mprotect`) – in case of OPP/MP the object/page is marked as dirty if the page faults on a write.

Page Buffer can be managed in many ways, with the simplest way being FIFO. Page Buffer pages are unprotected, so our user space implementation based runtime would have no information about how a page would be used while it remains in the Page Buffer, making LRU difficult to implement. For simplicity, we used FIFO in our current implementation. The only penalty is that if a dematerialized page is needed again then the page has to be rematerialized from RAM.

OPP can have more virtual memory usage than `malloc` for the same amount of data allocated. While MP will round each virtual address allocation to the next highest page size, the OPP model allocates one object per page. For 48-bit address spaces, the total number of pages is 2^{36} (≈ 64 Billion objects via OPP). For 32-bit systems, the corresponding number is 2^{20} (≈ 1 million objects). Programs that need to allocate more objects on 32-bit systems can use MP instead of OPP. Furthermore, SSDAlloc can coexist with standard `malloc`, so address space usage can be tuned by moving only necessary allocations to OPP.

While the separation between virtual memory and physical memory presents many avenues for DRAM optimization, it does not directly optimize SSD usage. We next present our SSD organization.

3.3 SSDAlloc’s SSD Maintenance

To overcome the limitations on random write behavior with SSDs, SSDAlloc writes the dirty objects when flushing the RAM object cache to the SSD in a log-structured [22] manner. This means that the objects have no fixed storage location on the SSD – similar to flash-based filesystems [11]. We first describe how we manage the mapping between fixed virtual address spaces to ever-changing log-structured SSD locations. Our SSD writer/garbage-collector is described later.

To locate objects on the SSD, SSDAlloc uses a data structure called the **Object Table**. While the virtual memory addresses of the objects are their fixed locations, Object Tables store their ever-changing SSD locations. Object Tables are similar to page tables in traditional virtual memory systems. Each Object Table has a unique identifier called the OTID and it contains an array of integers representing the SSD locations of the objects it indexes. An object’s Object Table Offset (OTO) is the offset in this array where its SSD location is stored. The 2-tuple $\langle \text{OTID}, \text{OTO} \rangle$ is the object’s internal persistent pointer.

To efficiently fetch the objects from the SSD when they are not cached in RAM, we keep a mapping between

each virtual address range (as allocated by the OPP or the MP memory manager) in use by the application and its corresponding Object Table, called an **Address Translation Module** (ATM). When the object of a page that is requested for materialization is not present in the RAM object cache, $\langle \text{OTID}, \text{OTO} \rangle$ of that object is determined from the page’s address via an ATM lookup (shown in Figure 1). Once the $\langle \text{OTID}, \text{OTO} \rangle$ is known, the object is fetched from the SSD, inserted into RAM object cache and the page is then materialized. The ATM is only used when the RAM object cache does not have the required objects. A successful lookup results in a materialized physical page that can be used without runtime system intervention for as long as the page resides in the Page Buffer. If the page that is requested does not belong to any allocated range, then the segmentation fault is a program error. In that case the control is returned to the originally installed seg-fault handler.

The ATM indexes and stores the 2-tuples $\langle \text{Virtual Memory Range}, \text{OTID} \rangle$ such that when it is queried with a virtual memory page address, it responds with the $\langle \text{OTID}, \text{OTO} \rangle$ of the object belonging to the page. In our implementation, we chose a balanced binary search tree for various reasons – 1) virtual memory range can be used as a key while the OTID can be used as a value. The search tree can be queried using an arbitrary page address and by using a binary search, one can determine the virtual memory range it belongs to. Using the queried page’s offset into this range, the relevant object’s OTO is determined, 2) it allows the virtual memory ranges to be of any size and 3) it provides a simple mechanism by which we can improve the lookup performance – by reducing the number of Object Tables, there by reducing the number of entries in the binary search tree. Our heap manager which allocates virtual memory (in OPP or MP style) always tries to keep the number of virtual memory ranges in use to a minimum to reduce the number of Object Tables in use. Before we describe our heap manager design, we present a few simple optimizations to reduce the size of Object Tables.

We try to store the Object Tables fully in DRAM to minimize multiple SSD accesses to read an object. We perform two important optimizations to reduce the size overhead from the Object Tables. First – to be able to index large SSDs for arbitrarily sized objects, one would need a 64 bit offset that would increase the DRAM overhead for storing Object Tables. Instead, we store a 32 bit offset to an aligned 512 byte SSD sector that contains the start of the object. While objects may cross the 512 byte sector boundaries, the first two bytes in each sector are used to store the offset to the start of the first object starting in that sector. Each object’s on-SSD metadata contains its size, using which, we can then find the rest of the object boundaries in that sector. We can index 2TB of

SSD this way. 40 bit offsets can be used for larger SSDs.

Our second optimization addresses Object Table overhead from small objects. For example, four byte objects can create 100% DRAM overhead from their Object Table offsets. To reduce this overhead, we introduce object batching – small objects are batched into larger contiguous objects. We batch enough objects together such that the size of the larger object is at least 128 bytes (restricting the Object Table overhead to a small fraction – $\frac{1}{32}$). Pages, however, are materialized in regular OPP style – one small object per page. However, batched objects are internally maintained as a single object.

3.4 SSDAlloc’s Heap Manager

Internally, SSDAlloc’s virtual memory allocation mechanism works like a memory manager over large Object Table allocations (shown in Figure 1). This ensures that a new Object Table is not created for every memory allocation. The Object Tables and their corresponding virtual memory ranges are created in bulk and memory managers allocate from these regions to increase ATM lookup efficiency. We provide two kinds of memory managers – An object pool allocator which is used for individual allocations and a `ptmalloc` style coalescing memory manager. We keep the pool allocator separate from the coalescing allocator for the following reasons: 1) Many of our focus applications prefer pool allocators, so providing a pool allocator further eases their development, 2) Pool allocators reduce the number of page reads/writes by not requiring coalescing, and 3) Pool allocators can export simpler memory usage information, increasing garbage collector efficiency.

Object Pool Allocator: SSDAlloc provides an object pool allocator for allocating objects individually via OPP. Unlike traditional pool allocators, we do not create pools for each object type, but instead create pools of different size ranges. For example, all objects of size less than 0.5KB are allocated from one pool, while objects with sizes between 0.5KB and 1KB are allocated from another pool. Such pools exist for every 0.5KB size range, since OPP performs virtual memory operations at page granularity. Despite the pools using size ranges, we avoid wasting space by obtaining the actual object size from the application at allocation time, and using this size both when the object is stored in the RAM object cache, and when the object is written to the SSD. When reading an object from the SSD, the read is rounded to the pool size to avoid multiple small reads.

SSDAlloc maintains each pool as a free list – a pool starts with a single allocation of 128 objects (one Object Table, with pages contiguous in virtual address space) initially and doubles in size when it runs out of space (with a single Object Table and a contiguous virtual memory range). No space in the RAM object cache or

the SSD is actually used when the size of pool is increased, since only virtual address space is allocated. The pool stops doubling in size when it reaches a size of 10,000 (configurable) and starts linearly increasing in steps of 10,000 from then on. The free-list state of an object can be used to determine if an object on the SSD is garbage, enabling object-granularity garbage collection. This type of a separation of the heap-manager state from where the data is actually stored is similar to the “frame-heap” implementation of Xerox Parc’s Mesa and Cedar languages [15].

Like Object Tables, we try to maintain free-lists in DRAM, so the free list size is tied to the number of free objects, instead of the total number of objects. To reduce the size of the free list we do the following: the free list actively indexes the state of only one Object Table of each pool at any point of time, while the allocation state for the rest of the Object Tables in each pool is managed using a compact bitmap notation along with a count of free objects in each Object Table. When the heap manager cannot allocate from the current one, it simply changes the current Object Table’s free list representation to a bitmap and moves on to the Object Table with the largest number of free objects, or it increases the size of the pool.

Coalescing Allocator: SSDAlloc’s coalescing memory manager works by using memory managers like `ptmalloc` [5] over large address spaces that have been reserved. In our implementation we use a simple *best-first with coalescing* memory manager [5] over large pre-allocated address spaces, in steps of 10,000 (configurable) pages; no DRAM or SSD space is used for these pre-allocations, since only virtual address space is reserved. Each object/page allocated as part of the coalescing memory manager is given extra metadata space in the header of a page to hold the memory manager information (objects are then appropriately offset). OPP arrays of any size can be allocated by performing coalescing at the page granularity, since OPP arrays are simply arrays of pages. MP pages are treated like pages in the traditional virtual memory system. The memory manager works exactly like traditional `malloc`, coalescing freely at byte granularity. Thus, MP with our *Coalescing Allocator* can be used as a drop-in replacement for log-structured swap.

A dirty object evicted by RAM object cache needs to be written to the SSD’s log and the new location has to be entered at its OTO. This means that the older location of the object has to be garbage collected. An OPP object on the SSD which is in a free-list also needs to be garbage collected. Since SSDs do not have the mechanical delays associated with a moving disk head, we can use a simpler garbage collector than the seek-optimized ones developed for disk-based log-structured file systems [22]. Our cleaner performs a “read-modify-write” operation

over the SSD sequentially – it reads any live objects at the head of the log, packs them together, and writes them along with flushed dirty objects from RAM.

3.5 SSDAlloc’s Garbage Collector

The SSDAlloc Garbage Collector (GC) activates whenever the RAM object cache has evicted enough number of dirty objects (as shown in Figure 1) to amortize the cost of writing to the SSD. We use a simple read-modify-write garbage collector, which reads enough partially-filled blocks (of configurable size, preferably large) at the head of the log to make space for the new writes. Each object on the SSD has its 2-tuple $\langle \text{OTID}, \text{OTO} \rangle$ and its size as the metadata, used to update the Object Table. This back pointer is also used to figure out if the object is garbage, by matching the location in the Object Table with the actual offset. To minimize the number of reads per iteration of the GC on the SSD, we maintain in RAM the amount of free space per 128KB block. These numbers can be updated whenever an object in an erase block is moved elsewhere (live object migration for compaction), when a new object is written to it (for writing out dirty objects) or when the object is moved to a free-list (object is “free”).

While the design so far focused on obtaining high performance from DRAM and flash in a hybrid setting, memory allocated via SSDAlloc is not non-volatile. We now present our durability framework to preserve application memory and state on the SSD.

3.6 SSDAlloc’s Durability Framework

SSDAlloc helps applications make their data persistent across reboots. Since SSDAlloc is designed to use much more SSD-backed memory than the RAM in the system, the runtime is expected to maintain the data persistent across reboots to avoid the loss of work.

SSDAlloc’s checkpointing is a way to cleanly shutdown an SSDAlloc based application while making objects and metadata persistent to be used across reboots. Objects can be made persistent by simply flushing all the dirty objects from RAM object cache to the SSD. The state of the heap-manager, however, needs more support to be made persistent. The bitmap style free list representation of the OPP pool allocator makes the heap-manager representation of individually allocated OPP objects easy to be serialized to the SSD. However, the heap-manager information as stored by a coalescing memory manager used by the OPP based array allocator and the MP based memory allocator would need a full scan of the data on the SSD to be regenerated after a reboot. Our current implementation provides durability only for the individually allocated OPP objects and we wish to provide durability for other types of SSDAlloc data in the future.

We provide durability for the heap-manager state of the individually allocated OPP objects by reserving a

known portion of the SSD for storing the corresponding Object Tables and the free list state (a bitmap). Since the maximum Object Table space to object size overhead ratio is $\frac{1}{32}$, we reserve slightly more than $\frac{1}{32}$ of the total SSD space (by using a file that occupies that much space) where the Object Tables and the free list state can be serialized for later use.

It should be possible to garbage collect dead objects across reboots. This is handled by making sure that our copy-and-compact garbage collector is always aware of all the OTIDs that are currently active within the SSDAlloc system. Any object with an unknown OTID is garbage collected. Additionally, any object with an OTID that is active is garbage collected only according to the criteria discussed in Section 3.5.

Virtual memory address ranges of each Object Table must be maintained across reboots, because checkpointed data might contain pointers to other checkpointed data. We store the virtual memory address range of each Object Table in the first object that this Object Table indexes. This object is written once at the time of creation of the Object Table and is not made available to the heap manager for allocation.

3.7 SSDAlloc’s Overhead

We observe that the overhead introduced by the SSDAlloc’s runtime mechanism is minor compared to the performance limits of today’s high-end SSDs. On a test machine with a 2.4 GHz quad-core processor, we benchmark the SSDAlloc’s runtime mechanism to arrive at that conclusion. To benchmark the latency overhead of the signal handling mechanism, we protect 200 Million pages and then measure the maximum seg-fault generation rate that can be attained. For measuring the ATM lookup latency, we build an ATM with a million entries and then measure the maximum lookup throughput that can be obtained. To benchmark the latency of an on-demand page materialization of an object from the RAM object cache to a page within the Page Buffer, we populate a page with random data and measure the latency. To benchmark the page dematerialization of a page from the Page Buffer to an object in the RAM object cache, we copy the contents of the page elsewhere, `madvise` the page as not needed and reprotect the page using `mprotect` and measure the total latency. To benchmark the latency of TLB misses (through L3) we use a CPU benchmarking tool, the Calibrator [2], by allocating 15GB of memory per core. Table 4 presents the results. Latencies of all the overheads clearly indicate that they would not be a bottleneck even for the high-end SSDs like the FusionIO IOXtreme drives, which can provide up to 250,000 IOPS. In fact, one would need 5 such SSDs for the SSDAlloc runtime to saturate the CPU.

The largest CPU overhead is from the signal han-

Overhead Source	Avg. Latency (μ sec)
TLB Miss (DRAM read)	0.014
ATM Lookups	0.046
Page Materialization	0.138
Page Dematerialization	0.172
Signal Handling	0.666
Combined Overhead	0.833

Table 4: SSDAlloc’s overheads are quite low, and place an upper limit of over 1 million operations per second using low-end server hardware. This request rate is much higher than even the higher-performance SSDs available today, and is higher than even what most server applications need from RAM.

dling mechanism, which is present only because of a user space implementation. With an in kernel implementation, the VM pager can be used to manage the Page Buffer, which would further reduce the CPU usage. We designed OPP for applications with high read randomness without much locality, because of which, using OPP will not greatly increase the number of TLB (through L3) misses. Hence, applications that are not bottlenecked by DRAM (but by CPU, network, storage capacity, power consumption or magnetic disk) can replace DRAM with high-end SSDs via SSDAlloc and reduce hardware expenditure and power costs. For example, Facebook’s memcache servers are bottlenecked by network parameters [3]; their peak performance of 200,000 tps per server can be easily obtained by using today’s high-end SSDs as RAM extension via SSDAlloc.

DRAM overhead created from the Object Tables is compensated by the performance gains. For example, a 300GB SSD would need 10GB and 300MB of space for Object Tables when using OPP and MP respectively for creating 128 byte objects. However, SSDAlloc’s random read/write performance when using OPP is 3.5 times better than when using MP (shown in Section 5). Additionally, for the same random write workload OPP generates 32 times less write traffic to the SSD when compared to MP and thereby increases the lifetime of the SSD. Additionally, with an in kernel implementation, either the page tables or the Object Tables will be used as they both serve the same purpose, further reducing the overhead of having the Object Tables in DRAM.

4 Implementation and the API

We have implemented our SSDAlloc prototype as a C++ library in roughly 10,000 lines of code. It currently supports SSD as the only form of flash memory, though it could later be expanded, if necessary, to support other forms of flash memory. In our current implementation, applications can coexist by creating multiple files on the SSD. Alternatively, an application can use the entire SSD, as a raw disk device for high performance. While the current implementation uses flash memory via an I/O

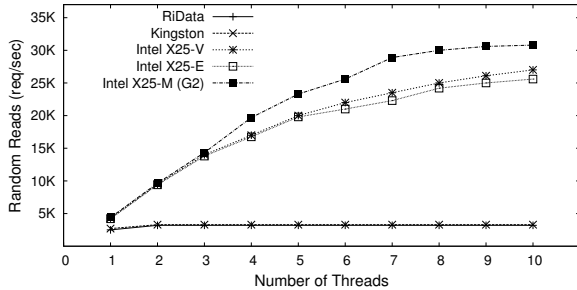


Figure 2: SSDAlloc’s thread-safe memory allocators allow applications to exploit the full parallelism of many SSDs, which can yield significant performance advantages. Shown here is the performance for 4KB reads.

controller such an overhead may be avoided in the future [13]. We present an overview of the implementation via a description of the API.

ssd_oalloc: *void* ssd_oalloc(int numObjects, int object-Size)*: is used for OPP allocations – both individual and array allocations. If *numObjects* is 1 then the object is allocated from the in-built OPP pool allocator. If it is more than 1, it is allocated from the OPP coalescing memory manager.

ssd_malloc: *void* ssd_malloc(size_t size)*: allocates *size* bytes of memory using the heap manager (described in Section 3.4) on MP pages. Similar calls exist for **ssd_calloc** and **ssd_realloc**.

ssd_free: *void ssd_free(void* va_address)*: deallocates the objects whose virtual allocation address is *va_address*. If the allocation was via the pool allocator then the $\langle OTID,OTO \rangle$ of the object is added to the appropriate free list. In case of array allocations, the in-built memory manager frees the data according to our heap manager. SSDAlloc is designed to work with low level programming languages like ‘C’. Hence, the onus of avoiding memory leaks and of freeing the data appropriately is on the application.

checkpoint: *int checkpoint(char* filename)*: flushes all dirty objects to the SSD and writes all the Object Tables and free-lists of the application to the file *filename*. This call is used to make the objects of an application durable.

restore: *int restore(char* filename)*: It restores the SSDAlloc state for the calling application. It reads the file (*filename*) containing the Object Tables and the free list state needed by the application and `mmaps` the necessary address for each Object Table (using the first object entry) and then inserts the mappings into the ATM as described in Section 3.6.

SSDs scale performance with parallelism. Figure 2 shows how some high-end SSDs have internal parallelism (for 0.5KB reads, other read sizes also have parallelism). Additionally, multiple SSDs could be used with

in an application. All SSDAlloc functions, including the heap manager, are implemented in a thread safe manner to be able to exploit the parallelism.

4.1 Migration to SSDAlloc

We believe that SSDAlloc is suited to the memory-intensive portions of server applications with minimal to no locality of reference, and that migration should not be difficult in most cases – our experience suggests that only a small number of data types are responsible for most of the memory usage in these applications. The following scenarios of migration are possible for such applications to embrace SSDAlloc:

- Replace all calls to `malloc` with `ssd_malloc`: Application would then use the SSD as a log-structured page store and use the DRAM as a page cache. Application’s performance would be better than when using the SSD via unmodified Linux swap because it would avoid random writes and circumvent other legacy swap system overheads that are more clearly quantified in FlashVM [23].
- Replace all `malloc` calls made to allocate memory intensive datastructures of the application with `ssd_malloc`: Application can then avoid SSDAlloc’s runtime intervention (copying data between Page Buffer and RAM object cache) for non-memory intensive datastructures and can thereby slightly reduce its CPU utilization.
- Replace all `malloc` calls made to allocate memory intensive datastructures of the application with `ssd_oalloc`: Application would then use the SSD as a log-structured object store only for memory intensive objects. Application’s performance would be better than when using the SSD as a log-structured swap because now the DRAM and the SSD would be managed at an object granularity.

In our evaluation of SSDAlloc, we tested all the above migration scenarios to estimate the methodology that provides the maximum benefit for applications in a hybrid DRAM/SSD setting.

5 Evaluation Results

In this section we evaluate SSDAlloc using microbenchmarks and applications built or modified to use SSDAlloc. We first present microbenchmarks to test the limits of benefits from using SSDAlloc versus SSD-swap. We also examine the performance of memcached (with SSDAlloc and SSD-swap), a popular key-value store used in datacenters, where SSDs have been shown to minimize energy consumption [7]. Later, we benchmark a B+Tree index for SSDs, where we replace all calls to `malloc` with `ssd_malloc` to see the benefits and impact of an automated migration to SSDAlloc.

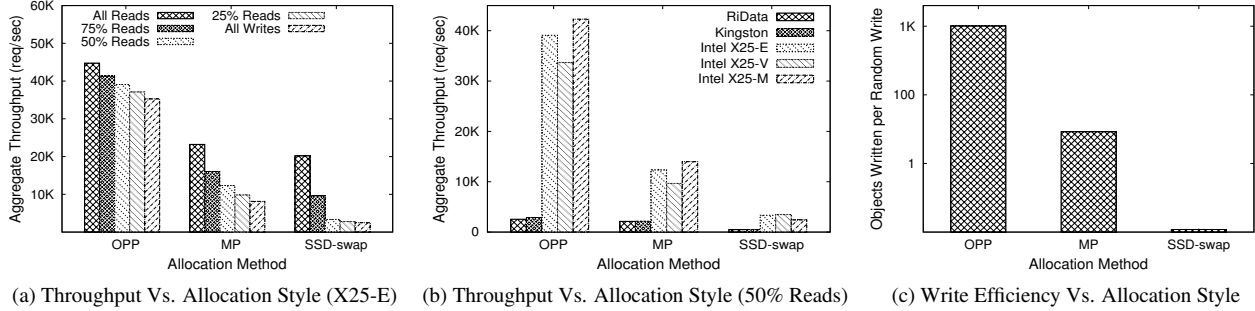


Figure 3: Microbenchmark results on 32GB object (128 byte each) array. In (a), OPP works best (1.8–3.5 times over MP and 2.2–14.5 times over swap), MP and swap take a huge performance hit when write traffic increases. In (b), OPP, on all SSDs, trumps all other methods by reducing read and write traffic. In (c), OPP has the maximum write efficiency (31.5 times over MP and 1013 times over swap) by writing only dirty objects as opposed to writing full pages containing them.

After that, we compare the performance of systems designed to use SSDAlloc to the same system specifically customized to use the SSD directly, to evaluate the overhead from SSDAlloc’s runtime. We examine a network packet cache backend that was built using transparent SSDAlloc techniques described in this paper and also the non-transparent mechanism described in our workshop paper [8]. We also evaluate the performance of a web proxy/WAN accelerator cache index for SSDs introduced in prior work [9, 8] and similar to the problems addressed more recently [6, 14]. Here, we demonstrate how using OPP makes efficient use of DRAM while providing high performance.

In all these experiments we evaluate applications using three different allocation methods: **SSD-swap** (via `malloc`), **MP** or log-structured SSD-swap (via `ssd_malloc`), **OPP** (via `ssd_oalloc`). Our evaluations use five kinds of SSDs and two types of servers. The SSDs and some of their performance characteristics are shown in Table 3. The two servers we use have a single core 2GHz CPU with 4GB of RAM and a quad-core 2.4GHz CPU with 16GB of RAM respectively.

5.1 Microbenchmarks

We examine the performance of random reads and writes in an SSD-augmented memory by accessing a large array of 128 byte objects – an array of total size of 32GB using various SSDs. We further restrict the accessible RAM in the system to 1.5GB to test out-of-DRAM performance. We access objects randomly (read or write) 2 million times per test. The array is allocated using four different methods – SSD-swap (via `malloc`), MP (via `ssd_malloc`), OPP (via `ssd_oalloc`). Object Tables for each of OPP, and MP occupy 1.1GB and 34MB respectively. Page Buffers are restricted to a size of 25 MB (it was sufficient to pin a page down while it was being accessed in an iteration). Remaining memory was used by the RAM object cache. To exploit the SSD’s parallelism, we run 8–10 threads that perform the random ac-

	OPP	MP	SSD-swap
Average (μsec)	257	468	624
Std Dev (μsec)	66	98	287

Table 5: Response times show that OPP performs best, since it can make the best use of the block-level performance of the SSD whereas MP provides page-level performance. SSD-swap performs poorly due to worse write behavior.

cesses in parallel.

The results of this microbenchmark are shown in Figure 3. Figure 3(a) shows how (for the Intel X25-E SSD) allocating objects via OPP achieves much higher performance. OPP beats MP by a factor of **1.8–3.5** times depending on the write percentage and it beats SSD-swap by a factor of **2.2–14.5** times. As the write traffic increases, MP and SSD-swap fare poorly due to reading/writing at a page granularity. OPP reads only 512 byte sector per object access as opposed to reading a 4KB page; it dirties only 128 bytes as opposed to dirtying 4KB per random write.

Figure 3(b) demonstrates how OPP performs better than all the allocation methods across all the SSDs when 50% of the operations are writes. OPP beats MP by a factor of **1.4–3.5** times and it beats SSD-swap by a factor of **5.5–17.4** times. Table 5 presents response time statistics when using the Intel X25-E SSD. OPP has the lowest averages and standard deviations. SSD-swap has a high average response time compared to OPP and MP. This is mainly because of storage sub-system inefficiencies and random writes (quantified more clearly in [23]).

Figure 3(c) quantifies the write optimization obtained by using OPP in log scale. OPP writes at an object granularity, which means that it can fit more number of dirty objects in a given write buffer when compared to MP. When a 128KB write buffer is used, OPP can fit nearly 1024 dirty objects in the write buffer while MP can fit only around 32 pages containing dirty objects. Hence, OPP writes more number of dirty objects to the SSD per random write when compared to both MP and SSD-

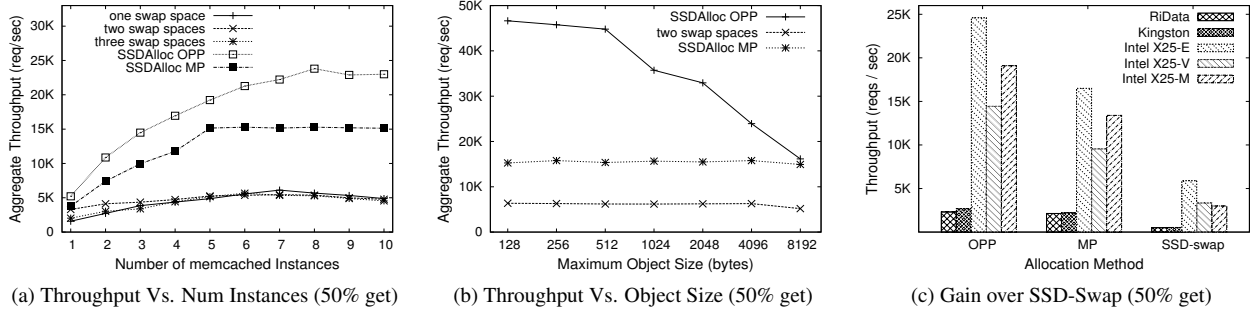


Figure 4: Memcached results. In (a), OPP outperforms MP and SSD-swap by factors of 1.6 and 5.1 respectively (mix of 4byte to 4KB objects). In (b), SSDAlloc’s use of objects internally can yield dramatic benefits, especially for smaller memcached objects. In (c), SSDAlloc beats SSD-Swap by a factor of 4.1 to 6.4 for memcached tests (mix of 4byte to 4KB objects).

swap (which makes a random write for every dirty object). OPP writes **1013** times more efficiently compared to SSD-swap and **31.5** times compared to MP (factors independent of SSD make). Additionally, OPP not only increases write efficiency but also writes **31.5** times less data compared to MP and SSD-swap for the same workload by working at an object granularity and thereby increases the SSD lifetime by the same factor.

Overall, OPP trumps SSD-swap by huge gain factors. It also outperforms MP by large factors providing a good insight into the benefits that OPP would provide over log-structured swaps. Such benefits scale inversely with the size of the object. For example with 1KB objects OPP beats MP by a factor of **1.6–2.8** and with 2KB objects the factor is **1.4–2.3**.

5.2 Memcached Benchmarks

To demonstrate the simplicity of SSDAlloc and its performance benefits for existing applications, we modify memcached. Memcached uses a custom slab allocator to allocate values and regular `mallocs` for keys. We replaced memcache’s slabs with OPP (`ssd_oalloc`) and with MP (`ssd_malloc`) to obtain two different versions. These changes require modifying 21 lines of code out of over 11,000 lines in the program. When using MP, we replaced `malloc` with `ssd_malloc` inside memcache’s slab allocator (used only for allocating values).

We compare these versions with an unmodified memcached using SSD-swap. For SSDs with parallelism we create multiple swap partitions on the same SSD. We also run multiple instances of memcached to exploit CPU and SSD parallelism. Figure 4 shows the results.

Figure 4(a) shows the aggregate throughput obtained using a 32GB Intel X25-E SSD (2.5GB RAM), while varying the number of memcached instances used. We compare five different configurations – memcached with OPP and MP, memcached with one, two and three swap partitions on the same SSD. For this experiment we populate memcached instances with object sizes distributed uniformly randomly from 4 bytes to 4KB such that the

total size of objects inserted is 30GB. For benchmarking, we generate 1 million memcached *get* and *set* requests (100% hitrate) each using four client machines that statically partition the keys and distribute their requests to all running memcached instances.

Results indicate that SSDAlloc’s write aggregation is able to exploit the device’s parallelism, while SSD-swap based memcached is restricted in performance, mainly due to the swap’s random write behavior. OPP (at 8 instances of memcached) beats MP (at 6 instances of memcached) and SSD-swap (at 6 instances of memcached on two swap partitions) by factors of 1.6 and 5.1 respectively by working at an object granularity, for a mix of object sizes from 4bytes to 4KB. While using SSD-Swap with two partitions lowers the standard deviation of the response time, SSD-Swap had much higher variance in general. For SSD-Swap, the average response time was 667 microseconds and the standard deviation was 398 microseconds, as opposed to OPP’s response times of 287 microseconds with a 112 microsecond standard deviation (high variance due to synchronous GC).

Figure 4(b) shows how object size determines memcached performance with and without OPP (Intel X25-E SSD). Here, we generate requests over the entire workload without much locality. We compare the aggregate throughput obtained while varying the maximum object size (actual sizes are distributed uniformly from 128 bytes to limit). We perform this experiment for three settings – 1) Eight memcached instances with OPP, 2) Six memcached instances with MP and 3) Six memcached instances with two swap partitions. We picked the number of instances from the best performing numbers obtained from the previous experiment. We notice that as the object size decreases, memcached with OPP performs much better than when compared to memcached with SSD-swap and MP. This is due to the fact that using OPP moves objects to/from the SSD, instead of pages, resulting in smaller reads and writes. The slight drop in performance in case of MP and SSD-swap when moving from 4KB object size limit to 8KB is because the runtime

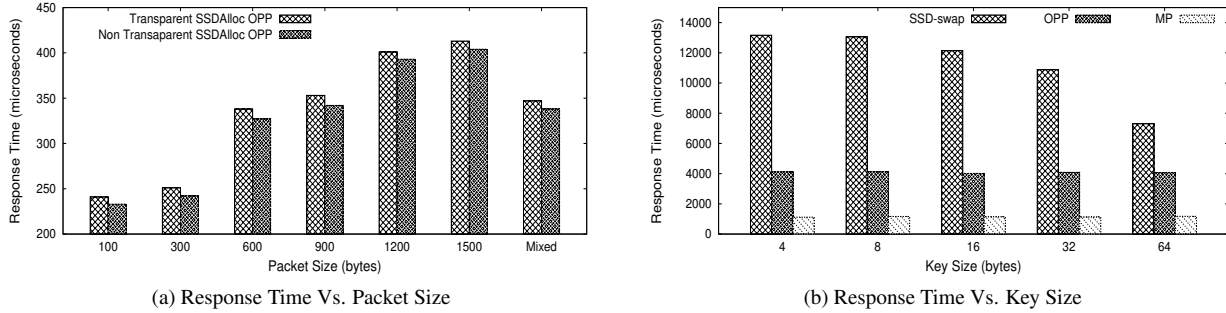


Figure 5: Packet Cache Benchmarks: In (a) we see that SSDAlloc’s runtime mechanism adds only up to 20 microseconds of latency overhead, while there was no significant difference in throughput. B+Tree Benchmarks: In (b), we see that SSDAlloc’s ability to internally use objects beats page-sized operations of MP or SSD-swap.

sometimes issues two reads for objects larger than 4KB. When the Object Table indicates that they are contiguous on SSD, we can fetch them together. In comparison, SSD-swap prefetches when possible.

Figure 4(c) quantifies these gains for various SSDs (objects between 4byte and 4KB) at a high insert rate of 50%. The benefits of OPP can be anywhere between **4.1–6.4** times higher than SSD-swap and **1.2–1.5** times higher than MP (log-structured swap). For smaller objects (each 0.5KB) the gains are **1.3–3.2** and **4.9–16.4** times respectively over MP and SSD-swap (16.4 factor improvement is achieved on the Intel X25-V SSD). Also, depending on object size distribution, OPP writes anywhere between **3.88–31.6** times more efficiently when compared to MP and **24.71–1007** times compared to SSD-swap (objects written per SSD write). The total write traffic of OPP is also between **3.88–31.6** times less when compared to MP and SSD-swap, increasing the lifetime and reliability of the SSD.

5.3 Packet Cache Benchmarks

Packet caches (and chunk caches) built using SSDs scale the performance of network accelerators [6] and inline data deduplicators [14] by exploiting good random read performance and large capacity of flash. Similar capacity DRAM-only systems will cost much more and also consume more power. We built a packet cache backend that indexes a packet with the SHA1 hash of its contents (using a hash table). We built it via two methods – 1) packets are allocated via OPP (`ssd_oalloc`), and 2) packets are allocated via the non-transparent object get/put based SSDAlloc that we describe in our workshop paper [8] – where the SSD is used directly without any runtime intervention. Remaining data structures in both the systems are allocated via `malloc`. We compare these two implementations to estimate the overhead from SSDAlloc’s runtime mechanism for each packet accessed.

For the comparison, we test the response times of packet get/put operations into the backend. We consider many settings – we vary the size of the packet from 100

to 1500 bytes and in another setting we consider a mix of packet sizes (uniformly, from 100 to 1500 bytes). We use a 20 byte SHA1 hash of the packet as the key that is stored in the hashtable (in DRAM) against the packet as the value (on SSD) – the cache is managed in LRU fashion. We generate random packet content from “/dev/random”. We use the Intel X25-M SSD and the high-end CPU machine for these experiments, with eight threads for exploiting device parallelism. We first fill the SSD with 32GB worth of packets and then perform 2 million lookups and inserts (after evicting older packets in LRU fashion). In this benchmark, we configured the Page Buffer to hold only a handful of packets such that every page get/put request leads to a signal raise, and an ATM lookup followed by an OPP page materialization.

Figure 5(a) compares the response times of OPP method using the transparent techniques described in this paper and the non-transparent calls described in the workshop paper [8]. The results indicate that the overhead from SSDAlloc’s runtime mechanism is only on the order of ten microseconds, there is no significant difference in throughput. Highest overhead observed was for 100 byte packets, where transparent SSDAlloc consumed 6.5% more CPU than the custom SSD usage approach when running at 38K 100 byte packets per second (30.4 Mbps). We believe this overhead is acceptable given the ease of development. We also built the packet cache by allocating packets via MP (`ssd_malloc`) and SSD-swap (`malloc`). We find that OPP based packet cache performed **1.3–2.3** times better than an MP based one and **4.8–10.1** times better than SSD-swap for mixed packets (from 100 to 1500 bytes) across all SSDs. Write efficiency of OPP scaled according to the packet size as opposed to MP and SSD-swap which always write a full page (either for writing a new packet or for editing the heap manager data by calling `ssd_free` or `free`). Using an OPP packet cache, three Intel SSDs can accelerate a 1Gbps link (1500 byte packets at 100% hit rate). Whereas, MP and SSD-swap would need 5 and 12 SSDs respectively.

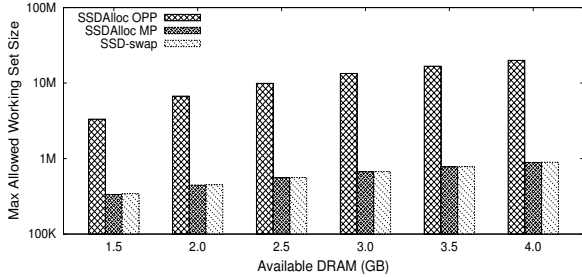


Figure 6: HashCache benchmarks: SSDAlloc OPP option can beat MP and SSD-Swap on RAM requirements due to caching objects instead of pages. The maximum size of a completely random working set of index entries each allocation method can cache in DRAM is shown (in log scale).

5.4 B+Tree Benchmarks

We built a B+Tree data structure via Boost framework [1] using the in-built Boost *object_pool* allocator (which uses `malloc` internally). We then ported it to SSDAlloc OPP (in 15 lines of code) by replacing calls to `object_pool` with `ssd_oalloc`. We also ported it to MP by replacing all calls to `malloc` (inside `object_pool`) with `ssd_malloc` (in 6 lines of code). Hence, in the MP version, every access to memory happens via the SSDAlloc’s runtime mechanism.

We use the Intel X25-V SSD (40GB) for the experiments and restrict the amount of memory in the system to 256MB for both the systems to test out-of-DRAM behavior. We allow up to 25 keys stored per inner node and 25 values stored in the leaf node, and we vary the key size. We first populate the B+Tree such that it has 200 million keys, to make sure that the height of the B+Tree is at least 5. We vary the size of the key, so that the size of the inner object and leaf node object vary. We perform 2 million *updates* (values are updated) and *lookups*.

Figure 5(b) shows that MP and OPP provide much higher performance than using SSD-swap. As the key size increases from 4 to 64 bytes, the size of the nodes increases from 216 bytes to 1812 bytes. The performance of SSD-swap and MP is constant in all cases (with MP performing **3.8** times better than SSD-swap with log-structured writes) because they access a full page for almost every node access, regardless of node size, increasing the size of the total dirty data, thereby performing more erasures on the SSD. OPP, in comparison, makes smaller reads when the node size is small and its performance scales with the key size in the B+Tree. We also report that across SSDs, B+Tree operations via OPP were **1.4–3.2** times faster when compared to MP and **4.3–12.7** times faster than when compared to SSD-swap (for a 64 byte key). In the next evaluation setting, we demonstrate how OPP makes the best use of DRAM transparently.

5.5 HashCache Benchmarks

Our final application benchmark is the efficient Web cache/WAN accelerator index based on HashCache [9]. HashCache is an efficient hash table representation that is devoid of pointers; it is a set-associative cache index with an array of sets, each containing the membership information of a certain (usually 8–16) number of elements currently residing in the cache. We wish to use an SSD backed index for performing HTTP caching and WAN Acceleration for developing regions. SSD backed indexes for WAN accelerators and data deduplicators are interesting because only flash can provide the necessary capacity and performance to store indexes for large workloads. A netbook with multiple external USB hard drives (upto a terabyte) can act as a caching server [8]. The inbuilt DRAM of 1–2 GB would not be enough to index a terabyte hard drive in memory, hence, we propose using SSDAlloc in those settings – the internal SSD can be used as a RAM supplement which can provide the necessary index lookup bandwidth needed for WAN Accelerators [16] which make many index lookups per HTTP object.

We create an SSD based HashCache index for 3 billion entries using 32GB SSD space. For creating the index, HashCache creates a large contiguous array of 128 byte sets. Each set can hold information for sixteen elements – hashes for testing membership, LRU usage information for cache maintenance and a four byte location of the cached object. We test three configurations of HashCache: with OPP (via `ssd_oalloc`), MP (via `ssd_malloc`) and SSD-swap (via `malloc`) to create the sets. In total, we had to modify 28 lines of code for these modifications. While using OPP we made use of *Checkpointing*. This is because we want to be able to quickly reboot the cache in case of power outages (netbooks have batteries and a graceful shutdown is possible in case of power outages).

Figure 6(a) shows, in log scale, the maximum number of useful index entries of a web workload (highly random) that can reside in RAM for each allocation method. With available DRAM varying from 2GB to 4.5GB, we show how OPP uses DRAM more efficiently than MP and SSD-swap. Even though OPP’s Object Table uses almost 1GB more DRAM than MP’s Object Table, OPP still is able to hold much larger working set of index entries. This is because OPP caches at set granularity while MP caches at a page granularity, and HashCache has almost no locality. Being able to hold the entire working set in memory is very important for the performance of a cache, since it not only saves write traffic but also improves the index response time.

We now present some reboot and recovery time measurements. Rebooting the version of HashCache built with OPP Checkpointing for a 32GB index (1.1GB Ob-

ject Table) took **17.66 sec** for the Kingston SSD (which has a sequential read speed of 70 MBPS).

We also report performance improvements from using OPP over MP and SSD-swap across SSDs. For SSDs with parallelism, we partition the index horizontally across multiple threads. The main observation is that using MP or SSD-swap would not only reduce performance but also undermine reliability by writing more number of times and more data to the SSD. OPP's performance is **5.3–17.1** times higher than when using SSD-Swap, and **1.3-3.3** times higher than when using MP across SSDs (50% insert rate).

6 Conclusion

SSDAlloc provides a hybrid memory management system that allows new and existing applications to easily use SSDs to extend the RAM in a system, while performing up to 17 times better than SSD-swap, up to 3.5 times better than log-structured SSD-swap and increasing the SSD's lifetime by a factor of up to 30 times with minimal code changes, limited to the memory allocation part of the application code. The performance of SSDAlloc based applications is close to that of custom-developed SSD applications. We demonstrate the benefits of SSDAlloc in a variety of contexts – a data center application (memcached), a B+Tree index, a packet cache backend and an efficient hashtable representation (HashCache), which required only minimal code changes, little application knowledge, and no expertise with the inner workings of SSDs.

7 Acknowledgments

We would like to thank our shepherd, Eddie Kohler, as well as the anonymous NSDI reviewers. This research was partially supported by the NSF Awards CNS-0615237, CNS-0916204 and CNS-0519829.

References

- [1] Boost, . <http://www.boost.org/>.
- [2] Calibrator, . <http://homepages.cwi.nl/~manegold/Calibrator/#6>.
- [3] Scaling Memcached at Facebook, . http://www.facebook.com/note.php?note_id=39391378919.
- [4] Memcached, . <http://www.danga.com/memcached/>.
- [5] ptmalloc, . <http://www.malloc.de/en/>.
- [6] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *Proc. 7th USENIX NSDI*, San Jose, CA, Apr. 2010.
- [7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.
- [8] A. Badam and V. S. Pai. Beating Netbooks into Servers: Making Some Computers More Equal Than Others. In *Proc. 3rd ACM Workshop on Networked Systems for Developing Regions (NSDR)*, BigSky, MO, 2009.
- [9] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. Hashcache: Cache storage for the next billion. In *Proc. 6th USENIX NSDI*, Boston, MA, Apr. 2009.
- [10] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proc. ASPLOS'92*, 1992.
- [11] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *Operating Systems Review*, 42(2): 88–93, 2007.
- [12] M. Castro, A. Adya, B. Liskov, and A. C. Myers. Hac: Hybrid adaptive caching for distributed storage systems. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malô, France, Oct. 1997.
- [13] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.
- [14] B. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.
- [15] P. V. der Linder. *Expert C Programming: Deep C Secrets*. Prentice Hall, Englewood Cliffs, N.J, 1994.
- [16] S. Ihm, K. Park, and V. S. Pai. Wide-area Network Acceleration for the Developing World. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.
- [17] T. Kgil and T. N. Mudge. Flashcache: A NAND flash memory file cache for low power web servers. In *Proc. of CASES'06*, 2006.
- [18] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A New Linux Swap System for Flash Memory Storage Devices. In *In ICCSA'09*, 2008.
- [19] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In *Proc. ACM SIGMOD*, Vancouver, BC, Canada, June 2008.
- [20] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proc. HotOS XII*, Monte Verita, Switzerland, May 2009.
- [21] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to ssds, analysis of tradeoffs. In *Proceedings of EuroSys'09*, 2009.
- [22] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.
- [23] M. Saxena and M. M. Swift. Flashvm: Virtual memory management on flash. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.
- [24] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient b-tree layer for flash-memory storage systems. In *Proceedings of RTCSA'04*, 2004.
- [25] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proc. 6th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 1994.