Bridging the Memory-Storage Gap

Anirudh Badam

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Vivek S. Pai

November 2012

Abstract

The Internet has become indispensable in the developing world. It has become an important tool for providing entertainment, for enabling and bettering human communication, for delivering effective education, for conducting research, for spreading news, and for organizing people to rally to various causes. However, today, only a third of the world's population has quality access to the Internet. The Internet has two primary hindrances for expansion. First, the high cost of network connectivity in developing regions and second, the high cost of establishing new data centers to reduce the load on the existing data centers. Fortunately, caches in various forms help address both these problems by storing reusable content near the clients. Despite their importance, today's caches are limited in their scale because of the trends in the evolution of current memory-storage technologies.

The widening gap between memory and storage is limiting the performance of applications like caches. The ever-increasing amount of data and the need to access more of it quickly have further magnified the gap. Limited DRAM capacity of servers makes it difficult to obtain good in-memory hit-rates that are vital for avoiding high-latency disk accesses. This dissertation presents two new complementary methods (HashCache and SSDAlloc) to bridge this gap for caches that affect the performance of many applications including HTTP Web Proxy caches, wide area network accelerators, content distribution networks, and file backup services. First, we will develop HashCache, a novel method to drastically improve the memory efficiency of caches. By reducing the amount of memory needed for caching by up to 20 times, it reduces the cache's total cost of ownership. While HashCache makes more effective use of limited RAM in a system, SSDAlloc introduces a tier of new memory technology like NAND-Flash between RAM and disk to further bridge the gap.

SSDAlloc removes the impediments to the integration of new high-capacity memory technologies like NAND-Flash into the memory hierarchy. SSDAlloc is a novel

memory manager that helps applications like HashCache tier data transparently and efficiently between DRAM and NAND-Flash. With only a few modifications to an existing application, restricted to the memory allocation portions of the code, one can reap the benefits of new memory technologies. Additionally, with SSDAlloc, applications can obtain 90% of the raw performance of NAND-Flash, while existing transparent tiering mechanisms deliver only 6-30% of that. Furthermore, by cleverly discerning application behavior, SSDAlloc writes up to 32 times less data to NAND-Flash when compared to similar existing mechanisms. This greatly increases the reliability of NAND-Flash that has a limited lifetime unlike DRAM.

Acknowledgements

"And what, Socrates, is the food of the soul? Surely, I said, knowledge is the food of the soul." – Plato.

The last six years of my life have been a great source of knowledge and wisdom. I wish to express my gratitude for everyone who made them special. I consider myself extremely fortunate to have been in the company of many fantastic researchers while enjoying the undying support of my family and friends. They have all collectively helped me in staying focused.

I would like to begin by thanking my advisor Vivek Pai for his teachings, advice and support through the years. His approach towards research, deep insight into various topics and the ability to quickly get down to the problem at hand have been a great source of knowledge and inspiration. In many places, this thesis will show his philosophy towards systems problems – "what is the single dominant issue in a problem setting and how to design an elegantly simple solution to tackle it?". I hope to continue using this philosophy in my research since it has always led me to great results. I would also like to thank Vivek for being extremely flexible and understanding towards some of my personal issues. Much of thesis would not have been possible otherwise.

Graduate level courses taught by Michael Freedman, Jennifer Rexford and Larry Peterson were extremely helpful in bootstrapping my doctoral journey. I vividly remember some of the great papers that Michael picked carefully for his students to read. His courses, Advanced Operating Systems, System support for Virtual Worlds, and Datacenter Systems have always helped me in keeping abreast with the greatest findings in many related areas of research. They have always inspired me to look at my research from other perspectives that would increase my clarity of thought. I feel fortunate for having the opportunity to interact with him.

Next, I would like to thank my mentors during internships. At HP Labs Princeton, I had the opportunity to work with Jack Brassil who first helped me understand the nuances of corporate research labs and how to go about solving problems in such a setting. I will cherish some of this advice as I will be moving to one such corporate research lab in the near future.

At Intel Labs Pittsburgh, I had the great opportunity of working with four terrific researchers – Dave Andersen, Michael Kaminsky, Dina Papagiannaki and Srini Seshan. Their jovial nature and expertise in a broad set of topics helped me in gaining sharp intuition into some new areas of research. I hope to interact with them again in the future.

At Fusion-io, I had company of David Nellans and Robert Wipfel who taught me how juggle between research and product development. I am certain that I would require these skills in the future when a product development opportunity presents itself that aligns closely with my research agenda.

My interest in science began early in my high school. I would like to thank Mrs. Jayashree, Mrs. Prafulla, Mr. Sharma, Mr. Ramaiah, Mr. Madhusudan, Mr. Koteshwar Rao, and Mr. Surendranath for their inspirational lectures on physics, chemistry and mathematics. I would also like to thank my mentors at IIT Madras during my undergraduate studies. The courses and advice offered by Sivaram Murthy, Narayana Swamy and G. Srinivasan put me on the right track of research. I am thankful to them for giving me the first taste of the benefits from research.

I sometimes jokingly call the department my second home. It is true at some level and I would attribute that to the friends who acted as the pseudo-family in the department. I would like to thank Rajsekar Manokaran, David Steurer, Aravindan Vijayaraghavan, Aditya Bhaskara, Arun Raman, Srinivas Narayana, Mortiz Hardt and Prakash Prabhu for making the department a fun place to be. I would also like to thank everyone who was involved in setting up the system's lab and everyone who

was involved in the maintenance of the coffee machines over the years. These friends and resources were indispensable in keeping my spirits up.

Life outside the department in Princeton was always fun in the company of Rajsekar, Aravindan, Aditya, Srinivas, Arun, Vijay Krishnamurthy, Ashwin Subramani, Narayanan Raghupathi and Ketra Kanodia. I would like to specially thank Vijay for his selfless help during my initial days in this country. His car was a single source of conveyance for many of us. I would also like to thank Mr. and Mrs. Ramesh Penmetcha for inviting me to their home for holidays and sending me back with loads of home cooked food.

I would like to thank my roommates over the years for being a family away from family. It was a pleasure living with Rajsekar, Vijay, Aravindan, Aditya, and Narayanan. I will miss all the help, the late night discussions, the cooking experiments, and the xbox gaming that kept me going through the years. You guys truly rock.

The admins at Princeton truly deserve my sincere gratitude. Over the years, Melissa Lawson has been a great guide who periodically helped me in dealing with administrative deadlines and procedures. I really appreciate what she does for the students. I would also like to thank Mitra Kelly for processing all my reimbursements on time. Csstaff has always lent a great helping hand in my research. I would like to thank Scott Karlin, Paul Lawson, Joseph Crouthamel, Chris Miller and Chris Tengi for their help and support through the six years. I also sincerely acknowledge the following NSF grants which funded my research: CNS-0615237, CNS-0519829, CNS-0520053, and CNS-0916204. I am also grateful to the university, Technology for Developing Regions Center at Princeton, the Gordon Wu Foundation and the Siebel Scholars Foundation for offering their support via fellowships during my stay at Princeton.

I would like to thank my friends who stuck with me through the years since my childhood. I would like to thank Nikhil Rangaraju, Rajanikar Tota, Sahiti Paleti, Deepak Anchala, Rahul Thota, Vishwakanth Malladi and the rest of the gang for their fun company, for giving such encouragement and for being supportive through the years.

In the spirit of saving the best for the last, I would like to thank my wonderful family now. It is to them that I dedicate this thesis. I am extremely fortunate to have been given such great parents. Their selfless love, unshakable faith in my abilities, and their willingness to always put me first laid the foundation for my successful career. I am grateful to them for being able to pamper me even while I was away from home. I would also like to thank my sister for always being the motherly figure in this country and providing me with wonderful vacations at her home through the last six years. I would also like to thank my brother-in-law for providing valuable advice on various personal and professional issues.

This thesis would not have been possible without the love and support of my wonderful wife Priyanka. I am very fortunate to have a wife who can put up with the tantrums and the ever changing temperament of a researcher whose mood is often guided by the status of his research. Her strength and perseverance in the face of difficulties have been a great source of inspiration and have helped me stay humble. Her unrelenting support, undying love for me, a magical ability to cheer me up and a heart big enough to look beyond my flaws were vital for the completion of this thesis.

To my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet plays an important role in many people's lives. In the developed world, it has become indispensable for providing entertainment [74, 113], for enabling and bettering human communication [39, 95], for delivering effective education [59, 69], for conducting research [1, 6], for spreading news [49, 105] and for organizing people behind various philanthropic and political causes [35, 48]. It is therefore necessary to facilitate the further growth of the Internet so that its benefits can be provided to the entire world.

Today, however, only about a third of the world's population has uninterrupted and high-quality access to the Internet and its vast resources [58]. There are two primary hindrances for the further scalability and penetration of the Internet. First, the current/projected cost of network connectivity in regions with limited or no-access to the Internet is high not only in relative currency but also in absolute terms. For example, Google sponsored, next-generation, satellite-based Internet in many developing countries is expected to cost $500/Mbps/month by 2013 [76], a cost that is two orders of magnitude higher than what the customers in the United States have to pay [57]. The high cost translates to fewer links with higher traffic compared to the developed world.

Second, the servers that power the Internet are predominantly located in the developed world [30]. This further increases the load on the constrained long-distance satellite and fiber links to the developed world that deliver the content to many developing countries. The high cost of establishing new servers [5] also means that existing servers will be overloaded when more users gain access to the Internet. Fortunately, Internet caches in various forms can help alleviate both these problems. Caches form an essential part of various Internet scale systems ranging from HTTP caches [96], content distribution networks [3], network accelerators [87] to file backup services [31]. They reduce the pressure not only on the network links, but also on the servers by caching reusable information closer to the client population.

Despite their importance, today's caches are limited in their scale because of the limitations of the traditional techniques used for caching and the trends in the evolution of traditional memory (DRAM) and storage (disk) technologies. In order to improve the Internet access across the world, we must overcome these limitations and create scalable and inexpensive caches. We must first rethink the caching techniques so that they make the best use of today's memory (DRAM) and storage (disk) technologies. Furthermore, we must investigate ways to incorporate new and efficient memory technologies like NAND-Flash memory (NAND-Flash) and Phase-change memory (PCM) into the design of caches to make them scale further.

1.1 Background and Motivation

Caches use local memory and storage to reduce redundant data fetches over the network. By caching information closer to the clients, they reduce network traffic and also the perceived latency on the network link to the server. By reducing the number of effective requests to the server they also reduce the load on the server.

Unfortunately, the current design of caches is tailored for deployment in developed countries. For example, with current caching data structures, the DRAM consumption is proportional to size of the disk used; also, the memory overhead per object cached is high. This stems from the fact that the index for the disk based cache, required to answer membership queries, is stored entirely in memory [28, 96] and is not optimized for memory constrained environments [100].

1.1.1 In-memory index based caches

Traditional cache indexes have high memory requirements. For example, the Squid open source HTTP proxy requires >80 bytes of DRAM per cached object [97] and cutting edge HTTP proxies require >32 bytes of DRAM per cached object [28]. Using these indexing techniques would require more than 10GB of DRAM for indexing only a terabyte of disk, assuming an average object size of 8KB.

Network accelerators require caches that are capable of indexing objects that are much smaller [54]. For example, using current techniques, a network accelerator configured to use chunks of 256 bytes for eliminating redundancy would require 320GB of DRAM for indexing a terabyte of disk. Moving forward, this would drive up the DRAM requirements of caches considering the trends in the content-based cacheability of content in the Internet [53]. It is, therefore, important to redesign the indexing data structures so that they use significantly less memory and reduce the cost of caches.

Large content distribution networks that split content into chunks to serve them individually also require large caches [75, 82]. File storage services also require large caches to speed up the process of deduplication that allows them to detect redundantly-stored content in a file server [34, 37]. Reducing the memory-storage coupling of caches can benefit the above applications by helping them scale.

1.1.2 Adopting new memory technologies

While significant benefits can be obtained from optimizing the indexing data structures, DRAM and disk have other problems which warrant the investigation of new memory technologies in designing caches.

Disk Property	**1982**	**2012**	**Change**
Maximum Drive Capacity	30 MB	3 TB	100,000x Better
Average Seek Latency	20 ms	7 ms	2.5x Better

Table 1.1: Changes in Disk Properties Over 30 Years.

Disks are not scaling. Table 1.1 demonstrates the poor scaling of disks over time. Even though the disks have been doubling their capacity fairly often to sustain the 100,000x growth in capacity in the last 30 years, their speeds have barely managed to increase by 3x. The high latency of disk accesses puts further pressure on the limited DRAM in servers. In comparison, today's high-performance NAND-Flash devices are capable of providing a million requests per second at a latency of under 100 microseconds with as much capacity as 10.24TB within a single rack unit [45] .

The limited number of seeks per disk means that high-performance servers have to rely heavily on DRAM for their performance. In fact, some high-performance services serve their content entirely out of DRAM and use disk purely for archival purposes [40, 51, 80]. However, DRAM has the following problems because of which such models of computing could face severe scalability challenges.

DRAM (GB)	**128**	**256**	**512**	**1024**	**2048**	**4096**
Space (RU)	6	6	10	40	80	80
Power (kW)	1.1	1.4	2.7	6.5	7.3	14.4

Table 1.2: DRAM density scaling in a Fujitsu SPARC Enterprise M series server.

DRAM does not scale up. Currently, 4TB of DRAM is the upper limit of DRAM that can fit in a single system image machine. Table 1.2 demonstrates how the Fujitsu SPARC Enterprise M series scales in space and power consumption with the total amount of DRAM contained in the system. 80 rack units of space is required to reach the maximum 4TB of DRAM. For comparison, a 1 rack unit server can hold 10TB of NAND-Flash.

DRAM is power hungry. The SuperMicro SuperServer 5086B-TRF (5 rack units) is one of the most DRAM dense servers on the market today and can pack 2TB of DRAM (1333 MHz ECC) into its 5U form factor. To power and cool this much DRAM, it draws 2.8kW of power. This same server can be alternatively outfitted with 45 TB of PCIe based NAND-flash within the same power budget and form factor. To scale DRAM to a comparable 44TB would require a cluster of 22 such machines that would consume an aggregate 110 rack units of space and 62kW of power. So, while possible to scale DRAM out, versus up, it is still far more effective to scale capacity using higher density technologies such as NAND-flash.

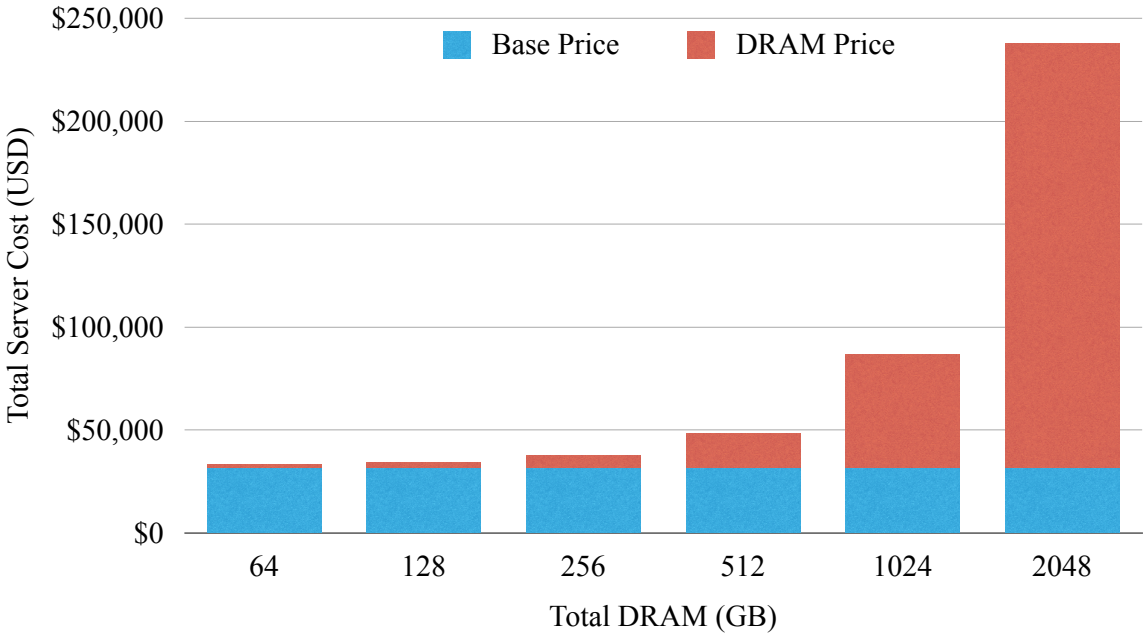

Figure 1.1: Relative costs when increasing DRAM capacity within a single server.

DRAM is expensive. Figure 1.1 shows the cost of a SuperMicro 5086B-TRF as the total DRAM is increased from 64GB to 2TB. Note super-linear cost increase in total server price as DRAM scales, due to the need for higher density DRAM parts. Building out a cluster with 44TB of DRAM would require approximately 5 million USD, or 8x more than it costs to equip a single server with 45TB of high end NAND-flash (assuming a cost of $12 USD/GB).

NAND-Flash, however, has problems of its own. Each NAND-Flash block can be modified only after an expensive operation called an *erase*. Furthermore, each NAND-Flash block is marked only for a limited number of erases after which the block cannot store data reliably. These limitations often need to be masked using software techniques [2, 52] that impose an obstacle to the quick adoption of NAND-Flash. These problems, however, are not specific to NAND-Flash but are a natural property of solid state storage technologies [63]. The above properties of DRAM, disk and NAND-Flash warrant the redesign of caches to help them make better use of current memory-storage technologies and also to help them incorporate new memory-storage technologies in an appropriate manner.

1.2 Our Approach and Contributions

To summarize, this dissertation attempts to answer the following questions:

1. How can one design a cache that can index disks that are larger by an order of magnitude when compared to existing designs without using any additional memory?

2. What are the performance implications of such a cache design? More specifically, can such a design match the performance of existing cache designs?

3. How must new memory technologies like NAND-Flash be used to reduce the reliance of such caches on traditional memory and disk for performance?

4. Can we adopt such new memory technologies transparently, while masking their limitations? More specifically, can applications use them without any modifications?

We address the first two questions in Chapter 2: we develop HashCache [11], a new way to design caches that drastically reduces the amount of memory needed to index a given amount of disk without sacrificing performance. In Chapter 3, we develop SSDAlloc [8, 10, 9] that helps applications like HashCache embrace new memory technologies like NAND-Flash in a transparent manner, while masking its limitations. Additionally, it increases performance by over an order of magnitude over existing transparent ways of using NAND-Flash.

1.2.1 Rethinking cache indexing for memory efficiency

We rethink the design of caches to increase their scalability in Chapter 2 by developing HashCache. HashCache makes the following contributions:

1. It is an efficient indexing mechanism which can index terabytes of disk with only a few megabytes of memory. It enables netbook-class machines to host caches with disks more than 2TB in size.

2. It helps caches use 20x less memory for the index when compared to Squid, while matching its performance.

3. It helps caches use 6x less memory for the index when compared to state-of-the-art high-performance caches, while matching their performance.

The approach behind HashCache was to start with a cache design that did not require any memory for indexing and then gradually increase the memory for the index to obtain higher performance. This approach helped us focus on a set of techniques that required the bare minimum memory for providing a certain level of

performance. Using the efficient indexing technique, the cache can either reduce its required memory and thereby, reduce the total cost of ownership or it can use the additional memory to speedup the accesses to the disk.

Chapter 2 presents eight different HashCache techniques that are closely related to each other. Each technique provides a different tradeoff between memory consumption and performance. The range of techniques provide flexibility in terms of the various practical settings where HashCache can be deployed. HashCache-SetMem can be deployed on a netbook-class machine and can provide the benefits of a cache to a small classroom in a developing country. HashCache-SetMem requires 20x less memory compared to Squid, while matching its performance.

HashCache-LogPrefetch can be deployed as a cache for an entire school in a developing country. HashCache-LogPrefetch requires 6x less memory compared to state-of-the-art high-performance caches, while matching their performance. For this reason, HashCache is equally suitable for deployment as a cache in the developed world.

1.2.2 Adopting new memory technologies

We develop SSDAlloc in Chapter 3 to help applications transparently embrace new memory technologies like NAND-Flash. It makes the following contributions:

1. SSDAlloc migrates new memory technologies like NAND-Flash transparently into the memory hierarchy of servers. This enables applications like HashCache to use NAND-Flash and not relying solely on DRAM. This helps applications scale not only their workload size, but also their performance.

2. SSDAlloc helps applications perform 3–17x better when compared to existing transparent techniques that help applications use NAND-Flash.

3. SSDAlloc helps applications obtain these performance benefits with only a few modifications to code, often restricted to the memory allocation portions. In

the applications that we have tested, we needed only 0.05% of lines of code to be modified to use SSDAlloc.

The approach behind SSDAlloc was to provide the benefits of customizing the application to exploit new memory technologies without actually having to rewrite the application. SSDAlloc abstracts out the core set of optimizations needed for adopting new memory technologies into the memory management layer of an OS. Most applications are rewritten to adopt new memory technologies like NAND-Flash by using them as a log-structured object based store [5, 4, 15, 26, 33, 107]. SSDAlloc cleverly abstracts such a management of NAND-Flash from the application by working under the virtual memory management sub-system of an OS.

Applications only need to modify their virtual memory management mechanism to embrace SSDAlloc. While simply replacing `malloc` with SSDAlloc can provide up to 17x times better performance over traditional memory management techniques, higher performance benefits can be obtained by NAND-Flash aware application redesign. However, with SSDAlloc one has the convenience of using NAND-Flash the way they would use DRAM. Therefore, the effort for redesigning the application would be reduced with SSDAlloc.

1.3 Dissertation Overview

This dissertation is organized in the following manner: in Chapter 1, we motivate the scalability problems of current caches and discuss how developing new cache designs and adopting new memory technologies will increase their scalability. We develop HashCache in Chapter 2, a new way to design caches that dramatically increases their capacity from a few tens of gigabytes to terabytes of disk without needing any extra memory. In Chapter 3, we develop SSDAlloc which is a technique to transparently

integrate new memory technologies like NAND-Flash into the memory hierarchy of an application. Finally, we conclude and discuss future work in Chapter 4.

Chapter 2

Redesigning Caching Techniques for Memory Efficiency

Network caching has been used in a variety of contexts to reduce network latency and bandwidth consumption, including FTP caching [90], Web caching [22, 23], redundant traffic elimination [72, 86, 87], and content distribution [3, 43, 82, 109]. All of these cases use local storage, typically disk, 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 [77] or the Classmate from Intel [56] 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, backhauled 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 2013 [76]. With efficient cache storage, one can reduce network connectivity expenses.

The goal of this chapter is to develop network cache stores designed for developing-world usage. In this chapter, 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 on par 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.

2.1 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. This reduces memory pressure inside systems that runs multiple applications like the serves in developing regions; (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 [36]. 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 can only index 30GB of disk space.

The rest of this chapter is structured as follows. Section 2.2 explains the current state of the art in network storage design. Section 2.3 explains the problem, explores a range of HashCache policies, and analyzes them. Section 2.4 describes our implementation of policies and the HashCache Web proxy. Section 2.5 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 2.6 describes the related work, Section 2.7 describes our current deployments, and Section 2.8 summarizes the chapter.

2.2 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 [23] 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 [22] proxy, by contrast, stored objects by creating directories that matched the components of the URL. By hashing the URL, Harvest was able to 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]

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 2.1: System Entities for Web Caches

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 any subsequent parts of the object 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) [88]. Unlike LFS, which is expected to retain files until deleted by the user, cache filesystems can often perform disk cache replacement in FIFO order, even if other approaches are used for main memory cache replacement. Table 2.1 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

[1] This 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.

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.

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 2.2: High Performance Cache - Memory Usage

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 2.2. Each entry has some object-specific information, such as its hash value and object size. It also has some disk-related 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 [24, 87, 94], 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.

2.3 Design

Policy	Bits/ Object	RAM GB/ 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 no seeks for misses
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 offset in memory
HC-LogLRU	15-47	0.23-0.67	$1 + \epsilon$	~ 0	0	log offset 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 2.3: 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

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 2.3.

2.3.1 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.

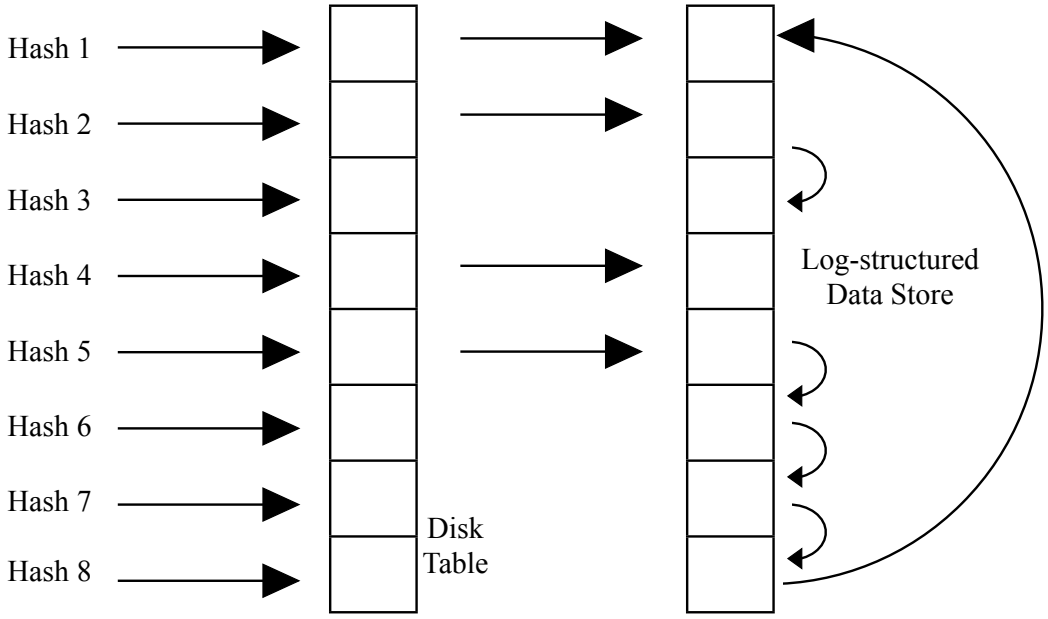

Figure 2.1: HashCache-Basic: objects with hash value i go to the ith bin for the first block of a file. Later blocks are in the circular log.

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, as illustrated in Figure 2.1.

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

2.3.2 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.

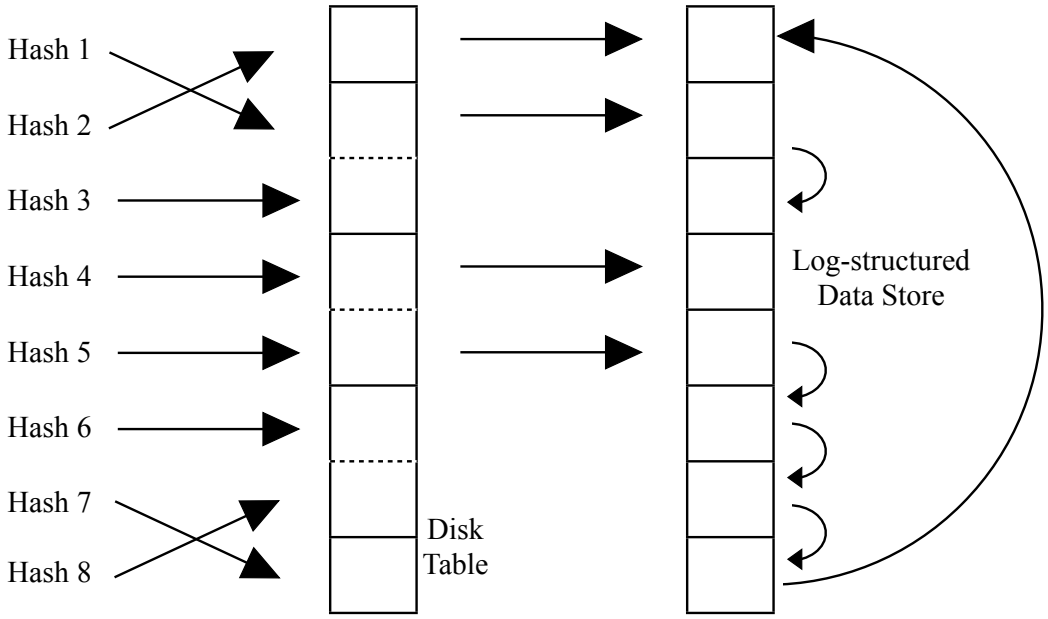

Figure 2.2: HashCache-Set: Objects with hash value i search through the $\frac{i}{N}^{th}$ set for the first block of a file. Later blocks are in the circular log. Some arrows are shown crossed to illustrate that objects that map on to a set can be placed anywhere in the set.

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 2.2, 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.

2.3.3 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 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 2.4.

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.

2.3.4 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 [36]. 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.

2.3.5 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 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 dissertation, we assume that all the policies have this optimization except HashCache-LogN which is the HashCache-Log policy without any prefetching.

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	$rr = \frac{t}{chr\cdot\left(1+\frac{1}{rel}\right)+(1-chr)\cdot cbr}$
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 2.4: Throughput 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.

2.3.6 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 2.4.

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 2.4 presents throughput 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. 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.

2.4 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.

2.4.1 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.

2.4.2 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 [81]. 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.

2.4.3 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 [28], 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.

2.4.4 Parameter Selection

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

Table 2.5: CDF of Web object sizes

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 2.5, we show an analysis of object sizes from a live, widely-used Web cache called CoDeeN [109]. 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.

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 2.6: Disk performance statistics

Since the choice of block size influences the set size, we make the decisions based on the performance of current disks. Table 2.6 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.

2.5 Performance Evaluation

In this section, we present experimental results that compare the performance of different indexing mechanisms presented in Section 2.3. 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 2.2 and is currently used in commercial service [106]. 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.

2.5.1 Workload

To evaluate these systems, we use the Web Polygraph [103] 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 [101] 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).

We use the latest standard workload, Polymix-4 [101], which was used at the Fourth Cache-off event [100] 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 [36].

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.

2.5.2 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 [78].

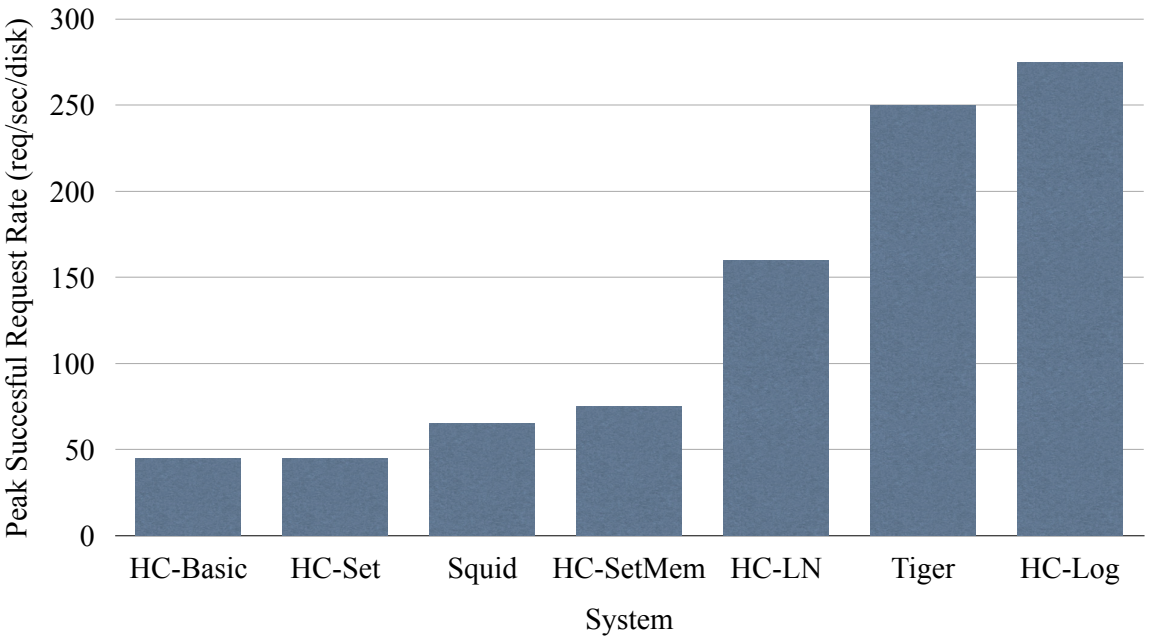

Figure 2.3: Peak Request Rates for Different policies for low end SATA disk.

Policy	**SATA Disk 7200 RPM**	**SCSI Disk 10000 RPM**	**SCSI Disk 15000 RPM**
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 2.7: Expected throughput (reqs/sec) for policies for different disk speeds– all calculations include read prefetching

Our tests for this machine configuration run at 40-275 requests per second, per disk, using either one or two disks. Figure 2.3 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 Squid do not suffer in performance as a result. Table 2.7 gives the analytical lower bounds 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 result is less than HashCache-Log because Tiger's larger index size reduces the amount of hot object cache available, reducing its prefetching effectiveness.

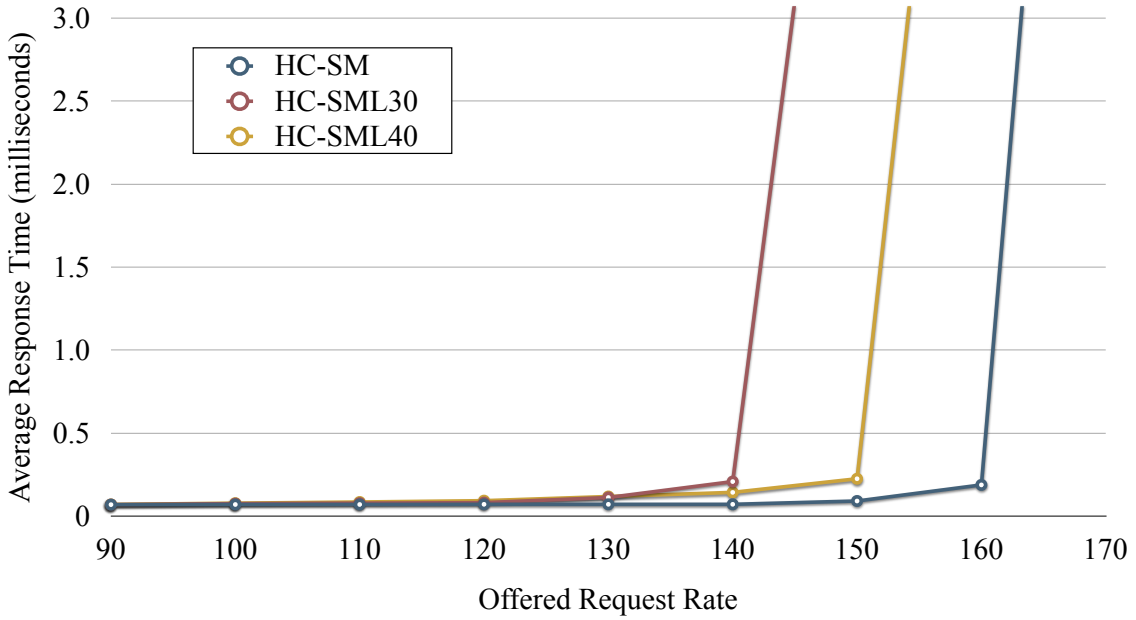

Figure 2.4: Peak Request Rates for Different SetMemLRU policies on low end SATA disks.

Figure 2.4 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 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.

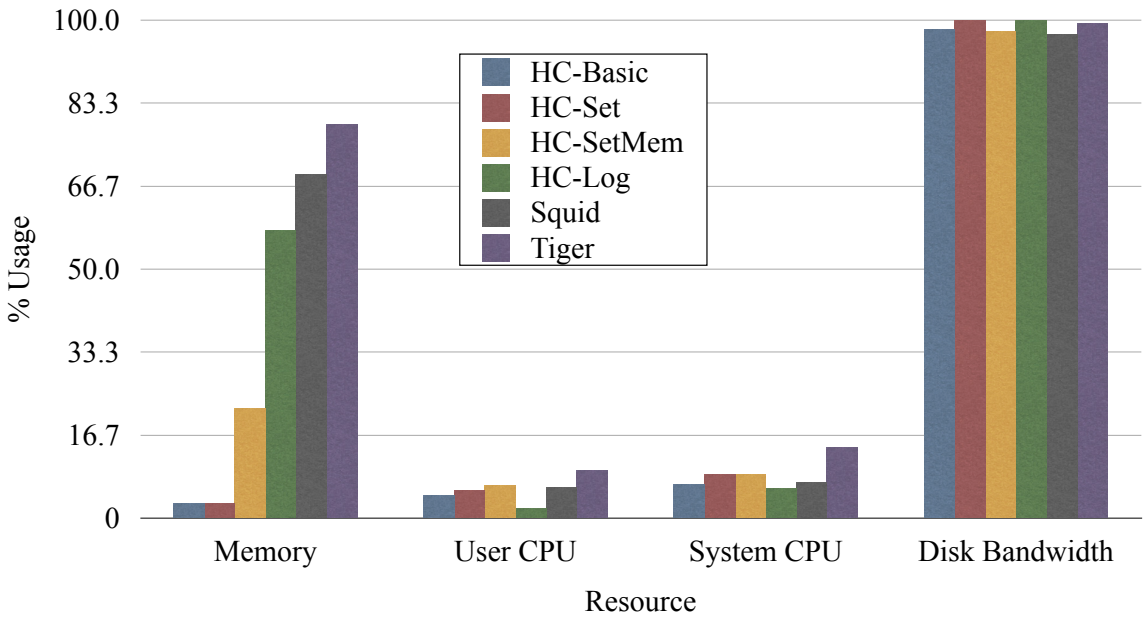

Figure 2.5: Resource Usage for Different Systems

These experiments also demonstrate HashCache's small systems footprint. Those measurements are shown in Figure 2.5 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. 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.

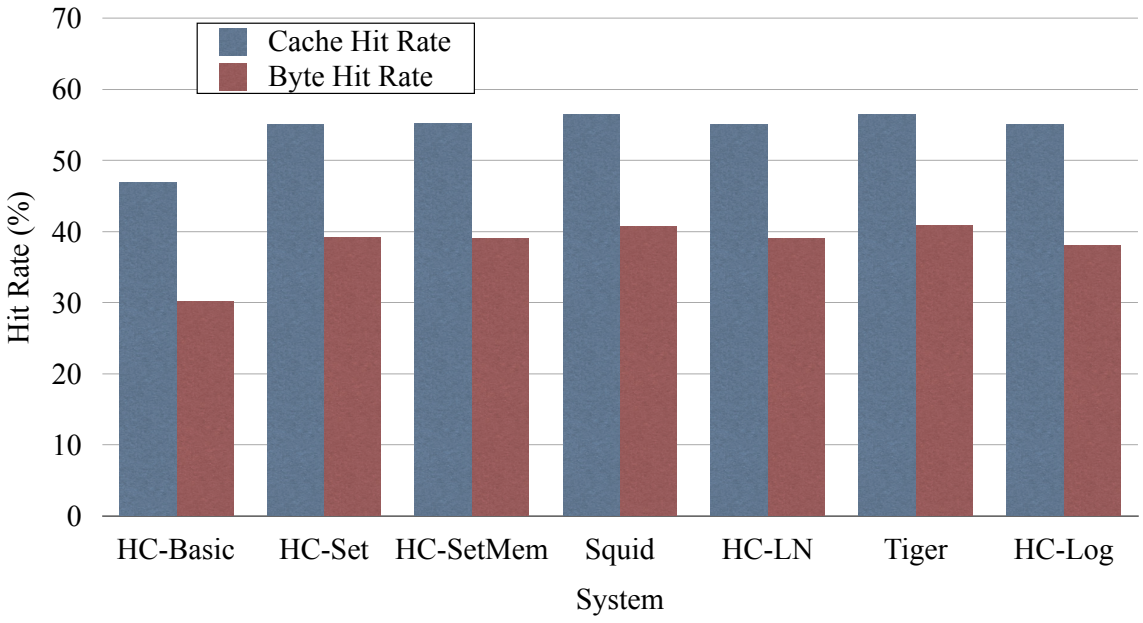

Figure 2.6: Low End Systems Hit Ratios

Figure 2.6 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.

2.5.3 High-End System Experiments

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

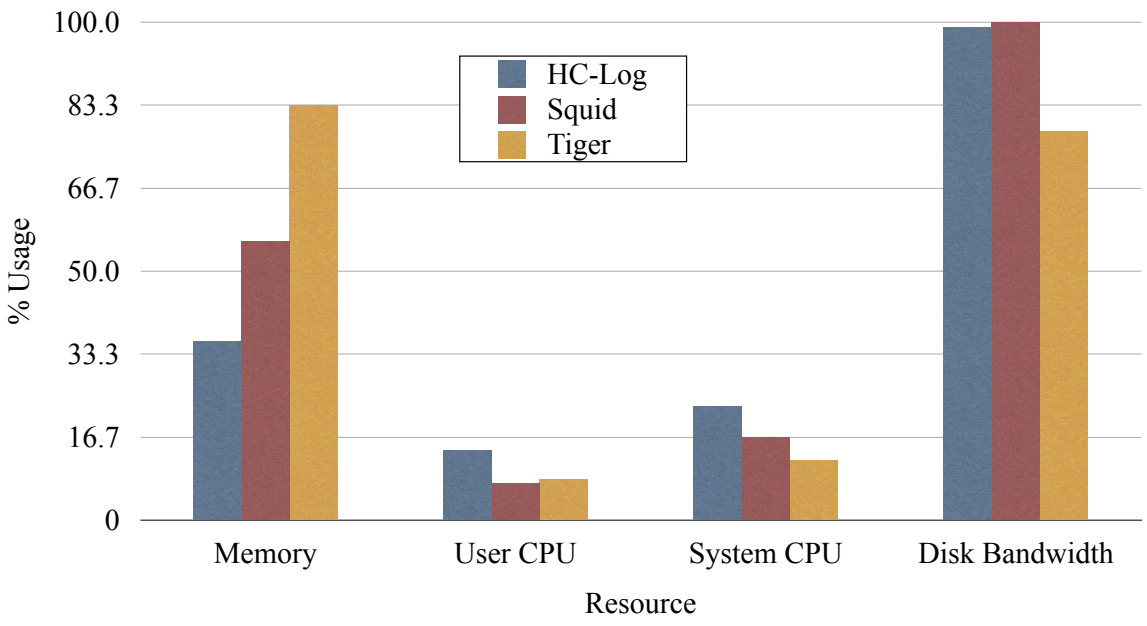

Figure 2.7: High End System Performance Statistics

Figure 2.7 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.

	TPS	BW Mb/s	Hit Time msec	All Time msec	Miss Time msec	CHR %	BHR %
HC-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 2.8: Performance on a high end system

The performance results from this test are shown in Table 2.8, 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 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.

2.5.4 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 of the configurations that have larger index requirements, representative of low-end servers being deployed [79].

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.

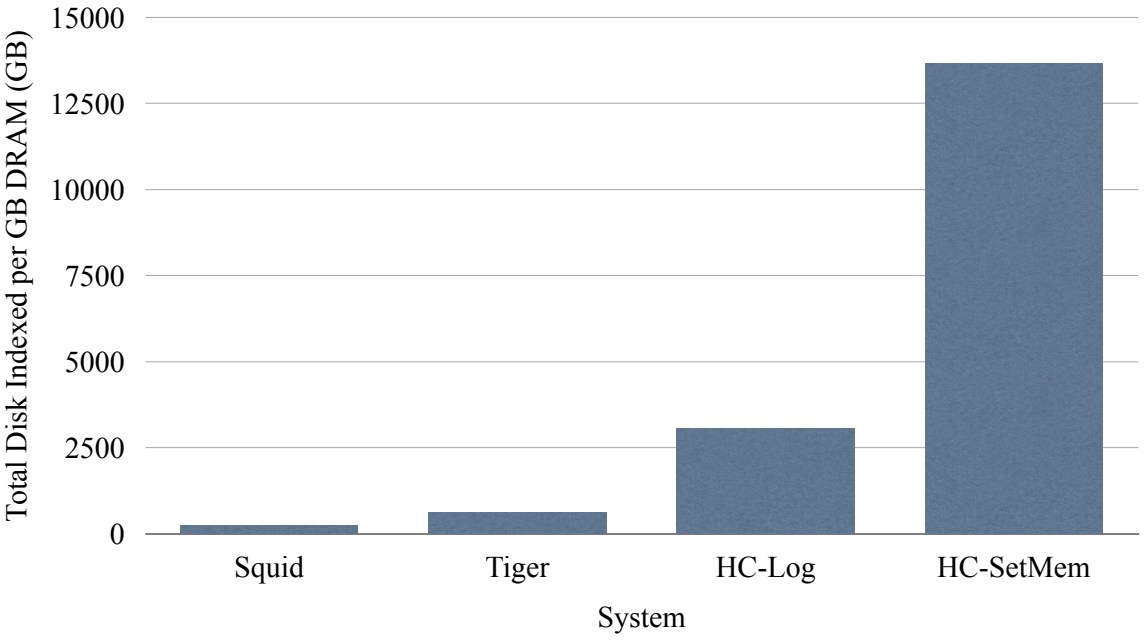

Figure 2.8: Sizes of disks that can be indexed by 2GB memory

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 2.8

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.

	TPS	**BW Mb/s**	**Hit Time msec**	**All Time msec**	**Miss Time msec**	**CHR %**	**BHR %**
HC-SetMem	75	3.96	27	1142	2508	57.12	40.11
HC-Log	300	16.02	48	1139	2507	57.88	40.21
HC-LogLRU	300	16.07	68	1158	2510	57.15	40.08

Table 2.9: Performance on 1TB disks

	TPS	**BW Mb/s**	**Hit Time msec**	**All Time msec**	**Miss Time msec**	**CHR %**	**BHR %**
HC-SetMem	150	7.98	32	1149	2511	57.89	40.89
HC-Log	600	32.46	56	1163	2504	57.01	40.07
HC-LogLRU	600	31.78	82	1171	2507	57.67	40.82

Table 2.10: Performance on 2TB disks

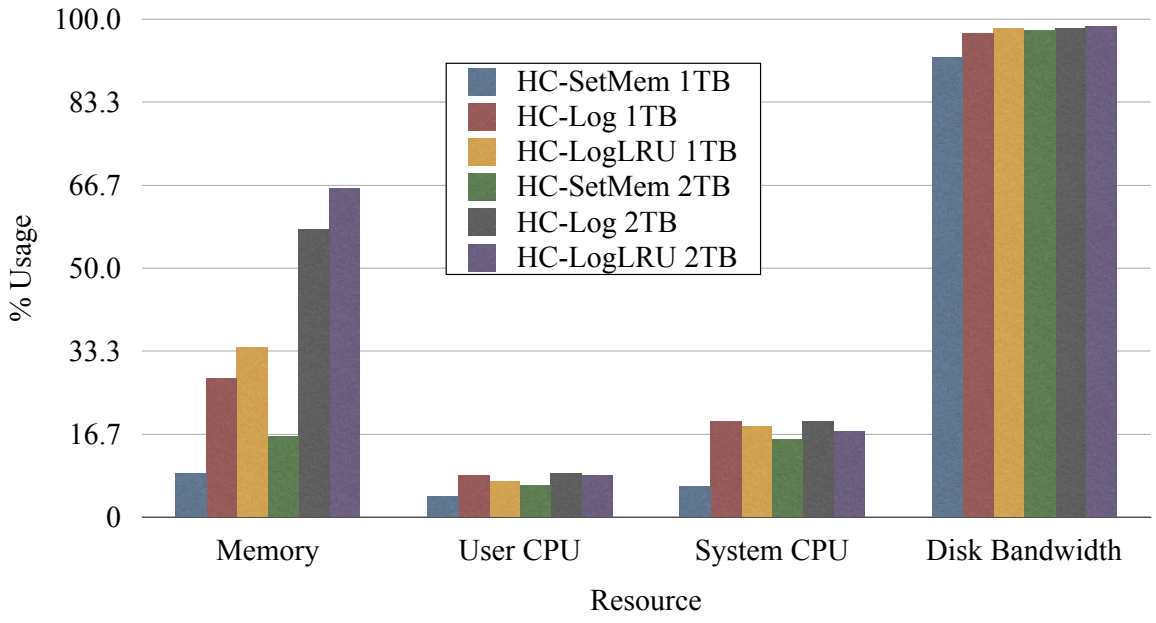

Figure 2.9: Large Disk System Performance Statistics

The Polygraph results for these configurations are shown in Tables 2.9, 2.10 , and the resource usage details are in Figure 2.9. 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 pre-loaded or shipped on DVDs later.

2.6 Related Work

Web caching in its various forms has been studied extensively in the research and commercial communities. As mentioned earlier, the Harvest cache [23] and CERN caches [66] were the early approaches. The Harvest design persisted, especially with its transformation into the widely-used Squid Web proxy [96]. Much research has been performed on Squid, typically aimed at reorganizing the filesystem layout to improve performance [66, 67], better caching algorithms [60], or better use of peer caches [47]. 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 [16] 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 [70, 92] 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 [19], proxy cache performance in mixed environments [42], and workload studies of enterprise user populations [50]. 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 [28], and one research project [93]. Based on published metrics, HashCache performs comparably to the commercial system, despite using a 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 publicly 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 [102] 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) [3, 43, 82, 109]. 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 geographi-

cally 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 SetMemLRU 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 [110] or rural WiFi [83, 98] 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.

2.7 Deployments

We deployed HashCache 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 ran on their university server which had a 100 GB hard drive, 2 GB memory and a dual core Xeon processor. For their Internet connection they paid $5,000 per month for a 2 Mbps satellite link to an ISP in Europe which had 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 was running other services such as a E-mail service and a firewall for the department and it was 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 ran on a wireless router for a small department on a 2 Mbps LAN. The Internet connection was through a 256 Kbps sub-marine link to Europe and the link had a ping latency ranging from 200 to 500 ms. The router had 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 were only 25 to 40 people using the router every day. Logging was disabled on this machine as well since we were asked not to consume network bandwidth for transferring the logs.

In both these deployments we have used HashCache policies to improve the Web performance while consuming a minimum amount of resources. 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.

2.8 Summary

We develop HashCache, a configurable indexing mechanism for caches. Using HashCache, caches can index terabytes of disk using only a few megabytes of memory. We develop eight different indexing policies that provide a different tradeoff between memory usage and performance. HashCache cache storage engine is 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 with positive results.

Chapter 3

Easing the Adoption of New Memory Technologies

An increasing number of networked systems today rely on in-memory (DRAM) indexes, hash tables, 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 128GB per server, jumping from a few thousand dollars to tens of thousands of dollars fairly quickly as shown in Figure 1.1. Power requirements scale similarly, restricting applications with large workloads from obtaining high in-memory hit-rates that are vital for high-performance.

NAND-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 compared to DRAM, and 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. The 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 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 chapter, 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, works at an object granularity for obtaining high performance and for maximizing the lifetime of flash memory. SSDAlloc's memory manager is compatible with 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 applications with benefits that are otherwise unavailable via existing transparent programming techniques.

Application	**Original LOC**	**Modified LOC**	**Throughput Gain vs**	
			Unmodified Swap	**Write-Logged Swap**
Memcache	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 3.1: 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.

Using SSDAlloc, we have modified four systems built originally using `malloc`: memcached [68] (a key-value store), a Boost [17] based B+Tree index, a packet cache back-end (for accelerating network links using packet level caching), and the HashCache [11] cache index. As shown in Table 3.1, 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 chapter is organized as follows: We describe related work and the motivation in Section 3.1. The design is described in Section 3.2, and we discuss our

implementation in Section 3.3. Section 3.4 provides the evaluation results, and we summarize the chapter in Section 3.5.

3.1 Motivation and Related Work

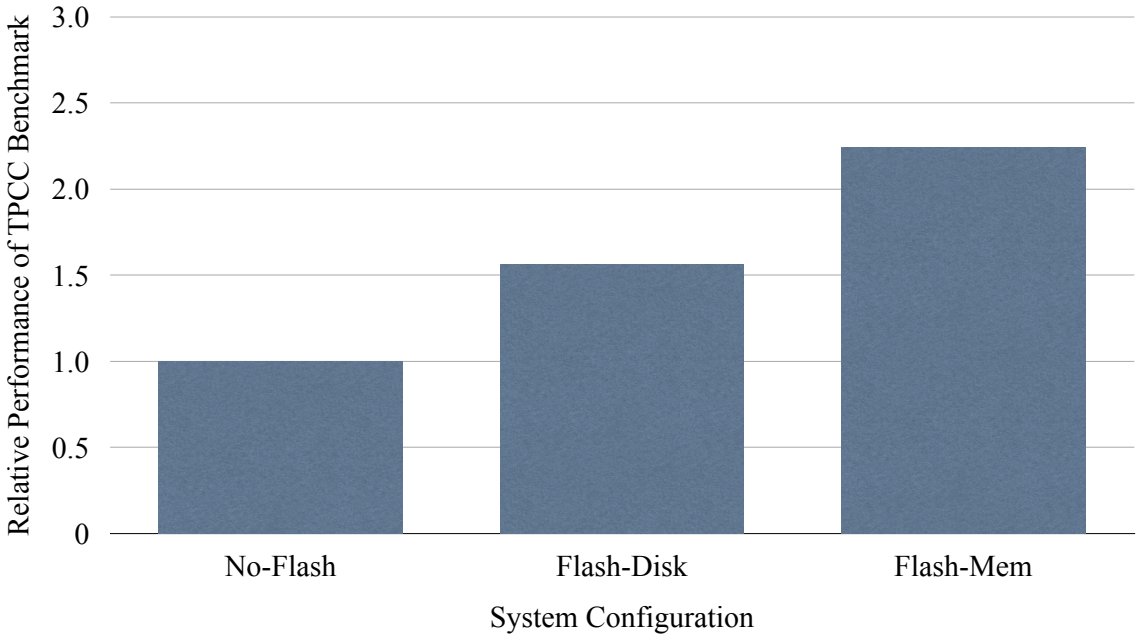

Figure 3.1: NAND-Flash can be better exploited when it is used as slow-memory as opposed to fast-disk

Figure 3.1 demonstrates why using SSDs as memory is better than using them as storage. We perform the following three experiments using MySQL InnoDB [55] transactional engine. In the first experiment (No-Flash), we configure a server (with a 2GHz quadcore processor, 48GB of DRAM and 512GB of disk) to run a TPCC benchmark [104] using a 450GB database. In the second experiment (Flash-Disk), we add a 80GB NAND-Flash SSD to the server. We configure the SSD to be used as block caching [38, 44] for the disk, and hence it is essentially being used as a storage device. In the final experiment (Flash-Mem), we use the NAND-Flash SSD as DRAM extension and the database transparently uses the SSD for its buffer pool.

Figure 3.1 shows the normalized performance of the three experimental setups. Flash-Disk performs 56% better than the No-Flash case. This is expected because the SSD performs much better when compared to disk. Flash-Mem performs 124% better than No-Flash case. The reasons behind the better performance of Flash-Mem are the following: First, traditional applications are written to take advantage of the low latency and high parallelism of DRAM, while they are optimized to avoid disk as much as possible. As a result, Flash-Mem based InnoDB is able to extract the benefits of low latency and high parallelism of SSDs. Second, traditional operating systems are designed to add multiple layers of buffering and reordering of requests for the storage sub-system, while they are explicitly optimized to get out of the way of memory accesses. As a result, Flash-Mem based InnoDB is able to circumvent the overhead from the think software layer between the application and the storage devices. Therefore, we propose the usage of SSDs as slow-memory and not as fast-disks [99].

While alternative memory technologies have been championed for more than a decade [12, 112], 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 chapter 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 [62, 71, 91] 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 [73].

FlashVM [91] 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) [64, 111] and Web servers [61] to indexes [4, 8] and key-value stores [5]. 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 chapter 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. A similar approach was used in distributed object systems [21], 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.

3.2 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 virtual memory.

- To utilize the DRAM in the system as efficiently as possible – Since most of the applications that we consider 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 3.2.1 and 3.2.4, 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 subsequent usage.

In Section 3.2.2, 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 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 3.2.3 and 3.2.5 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.

	Write Logging	Access < a page	Efficient GC	No DRAM Pollution	Retains Data	High Perf.	Familiar Interface
SSD Swap							✔
SSD Swap (Logged)	✔						✔
SSD `mmap`					✔		✔
App. Rewrite	✔	✔	✔	✔	✔	✔	
SSDAlloc	✔	✔	✔	✔	✔	✔	✔

Table 3.2: 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.

Table 3.2 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.

3.2.1 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 accesses 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 determines 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, in 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 (described in Section 3.2.2) 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 3.2.4). OPP is suitable for individual object allocations, typical of the applications we consider. OPP objects are stored on the SSD in a log-structured manner (details are explained in Section 3.2.5). 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 3.2.4 how an OPP-based coalescing allocator can be used to allocate OPP-based arrays.

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` [84] style coalescing memory manager (further explained in Section 3.2.4) 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 chapter, 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.

3.2.2 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.

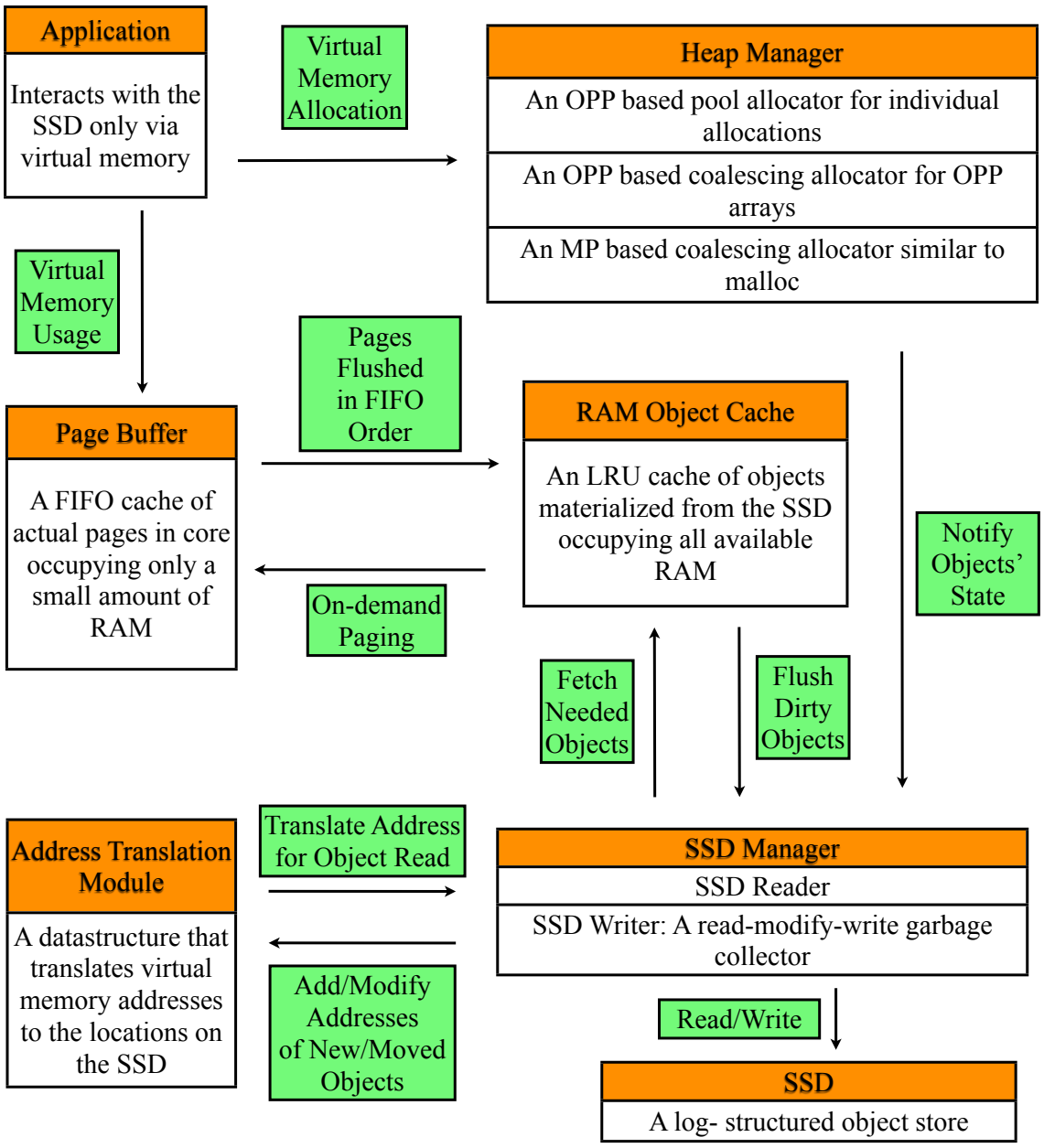

Figure 3.2: SSDAlloc uses most of RAM as an object-level cache, and materializes/dematerializes pages as needed to satisfy the application's page usage. This approach improves RAM utilization, even though many objects will be spread across a greater range of virtual address space.

RAM Object Cache – Objects are cached in the *RAM object cache* in a compact manner. The RAM object cache occupies most of the available portion of DRAM, while only a small part of DRAM is used for pages that are currently in use (shown

in Figure 3.2). 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.

The RAM object cache is maintained in an LRU fashion. It indexes objects using their virtual memory page address as the key. An OPP object in the 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 hash table 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 3.2).

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 reduce 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. The 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`). 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 the 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 the RAM object cache or the SSD.

Pages dematerialized from the Page Buffer are converted to objects. Those objects are pushed into the RAM object cache, the page is then `madvise`d 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.

The 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 use 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} ($\approx$ 64 Billion objects via OPP). For 32-bit systems, the corresponding number is 2^{20} ($\approx$ 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.

3.2.3 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 [88] manner. This means that the objects have no fixed storage location on the SSD – similar to flash-based filesystems [15]. 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 <OTID, OTO> 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, <OTID,OTO> of that object is determined from the page's address via an ATM lookup (shown in Figure 3.2). Once the <OTID,OTO> is known, the object is fetched from the SSD, inserted into the 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 <Virtual Memory Range, OTID> such that when it is queried with a virtual memory page address, it responds with the <OTID,OTO> 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.

3.2.4 SSDAlloc's Heap Manager

Internally, SSDAlloc's virtual memory allocation mechanism works like a memory manager over large Object Table allocations (shown in Figure 3.2). 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 ex-

ample, 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 a 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 [65].

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 [84] over large address spaces that have been reserved. In our implementation we use a simple *best-first with coalescing* memory manager [84] 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 the 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 [88]. 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.

3.2.5 SSDAlloc's Garbage Collector

The SSDAlloc Garbage Collector (GC) activates whenever the RAM object cache has evicted enough dirty objects (as shown in Figure 3.2) 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 <OTID,OTO> 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.

3.2.6 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 the

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's 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 3.2.5.

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.

3.2.7 SSDAlloc's Overhead

Overhead Source	**Avg. Latency (μsec)**
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 3.3: 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.

We observe that the overhead introduced by 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 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 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 [29], by allocating 15GB of memory per core. Table 3.3 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 handling 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 [40]; 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 offset 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 2.5). 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.

3.3 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 controller such an overhead may be avoided in the future [27]. We present an overview of the implementation via a description of the API.

ssd_oalloc: *void* ssd_oalloc(int numObjects, int objectSize)*: 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 3.2.4) 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 <OTID,OTO> 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 `mmap`s the necessary address for each Object Table (using the first object entry) and then inserts the mappings into the ATM as described in Section 3.2.6.

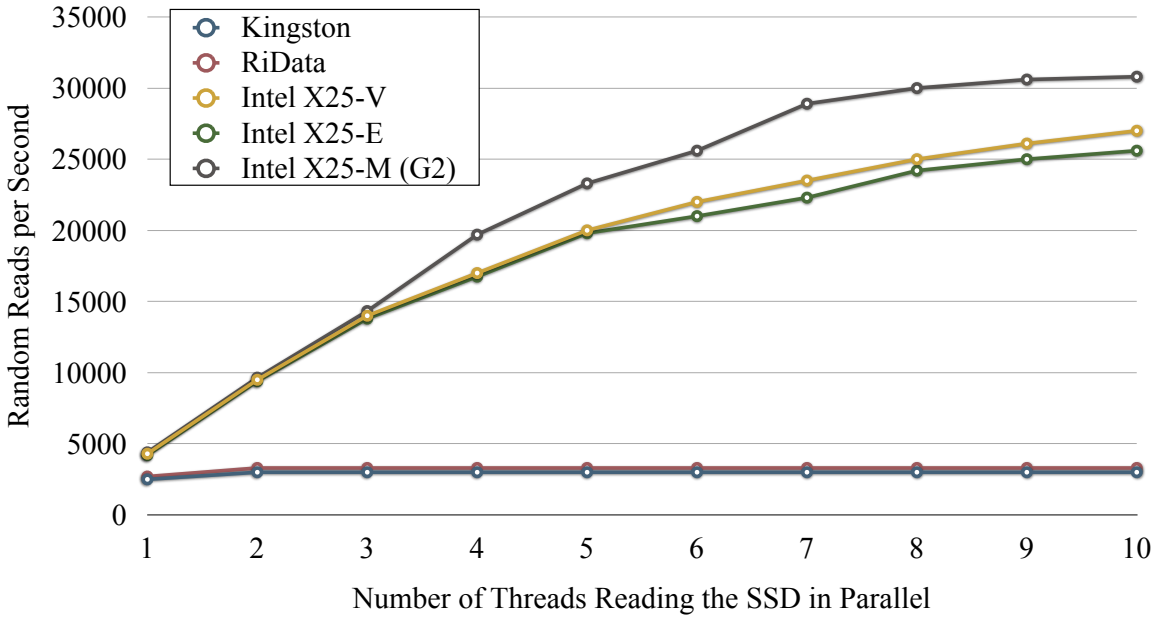

Figure 3.3: SSDAlloc's thread-safe memory allocators allow applications to exploit the full parallelism of many SSDs, which can yield significant performance advantages. Shown here is the performance for 4KB reads.

SSDs scale performance with parallelism. Figure 3.3 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.

3.3.1 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 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 quantified by Saxena et al [91].
- Replace all `malloc` calls made to allocate memory intensive data structures 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 data structures and can thereby slightly reduce its CPU utilization.
- Replace all `malloc` calls made to allocate memory intensive data structures 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.

3.4 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 data centers, where SSDs have been shown to minimize energy consumption [5]. 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.

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 back-end that was built using transparent SSDAlloc techniques described in this chapter 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 [11, 8] and similar to the problems addressed more recently [4, 32]. Here, we demonstrate how using OPP makes efficient use of DRAM, 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 3.4: SSDAlloc can take full advantage of object-sized accesses to the SSD, which can often provide significant performance gains over page-sized operations.

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`), and **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 3.4. 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.

3.4.1 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 is sufficient to pin a page down, while it is being accessed in an iteration). The remaining memory is used by the RAM object cache. To exploit the SSD's parallelism, we run 8–10 threads that perform the random accesses in parallel.

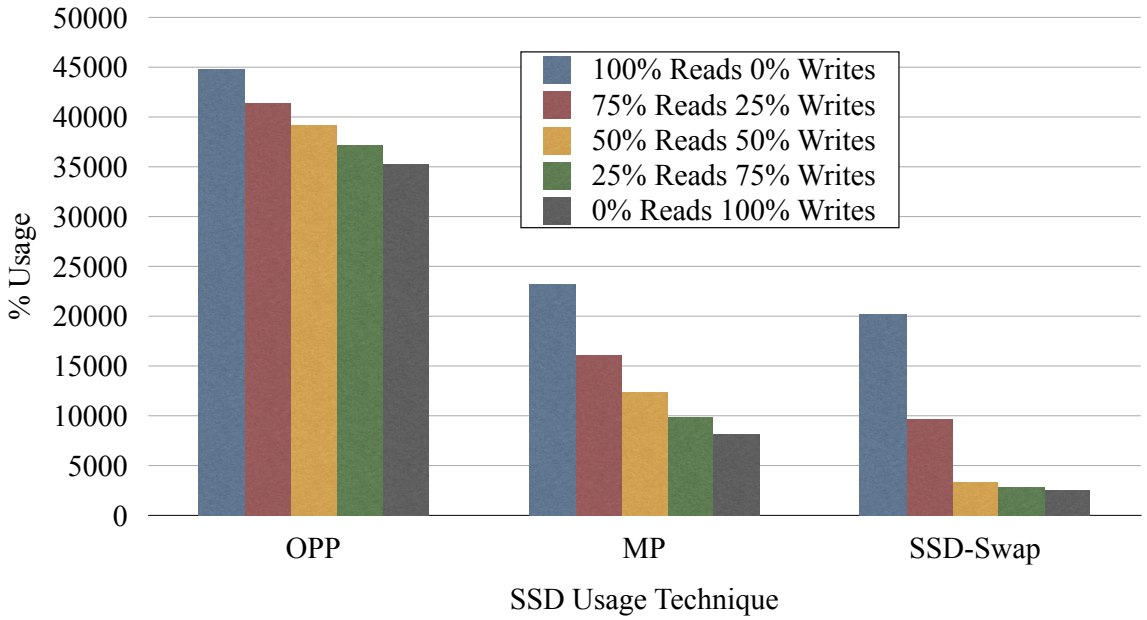

Figure 3.4: OPP works best (1.8–3.5 times over MP and 2.2–14.5 times over swap), MP and swap take a huge performance hit when write traffic increases

The results of these microbenchmarks are shown in Figures 3.4, 3.5, and 3.6. Figure 3.4 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.

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

Table 3.5: 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.

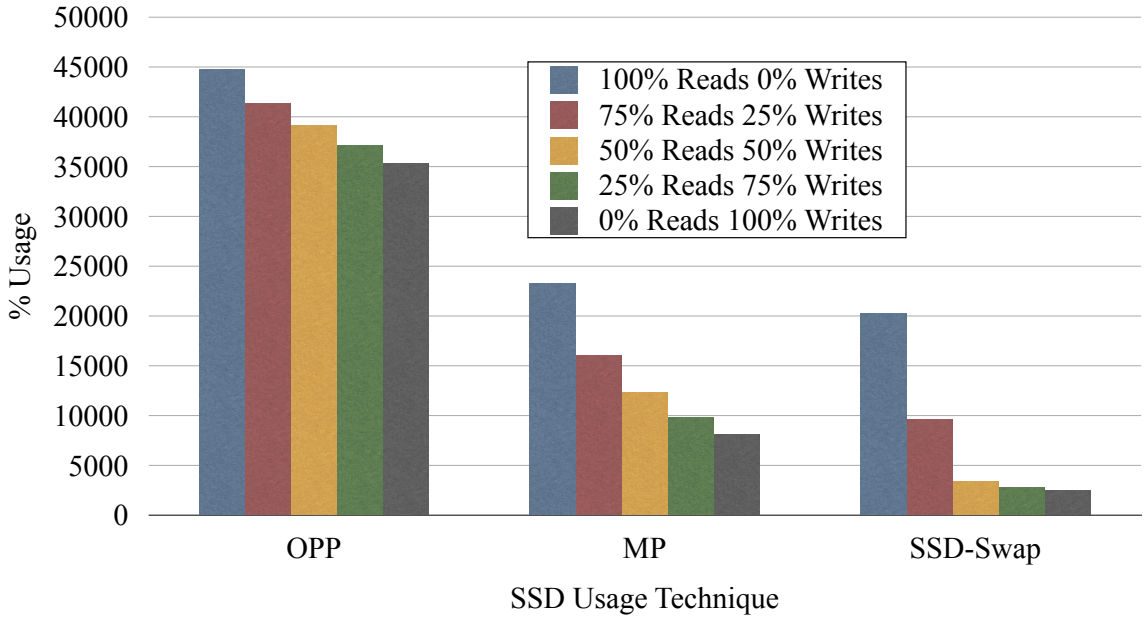

Figure 3.5: OPP, on all SSDs, trumps all other methods by reducing read and write traffic

Figure 3.5 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 3.5 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.

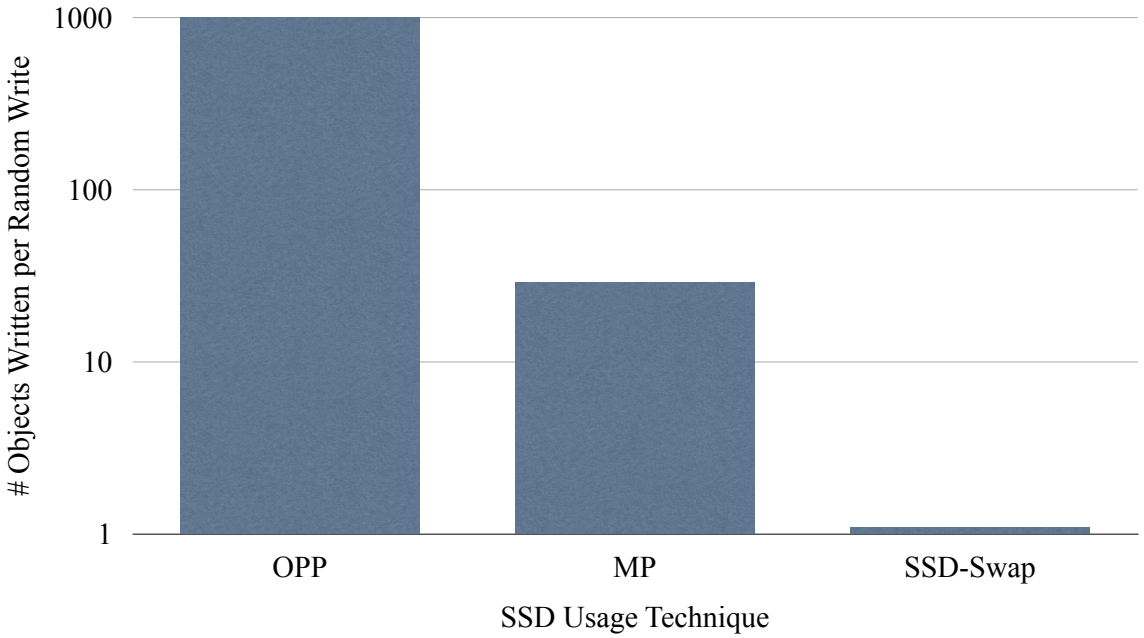

Figure 3.6: OPP has the maximum write efficiency (31.5 times over MP and 1013 times over swap) by writing only dirty objects as opposed to writing full pages containing them

Figure 3.6 quantifies the write optimization obtained by using OPP, in log scale. OPP writes at an object granularity, which means that it can fit more 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 dirty objects to the SSD per random write when compared to both MP and SSD-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**.

3.4.2 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 `malloc`s 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. Figures 3.7, 3.8, and 3.9 show the results.

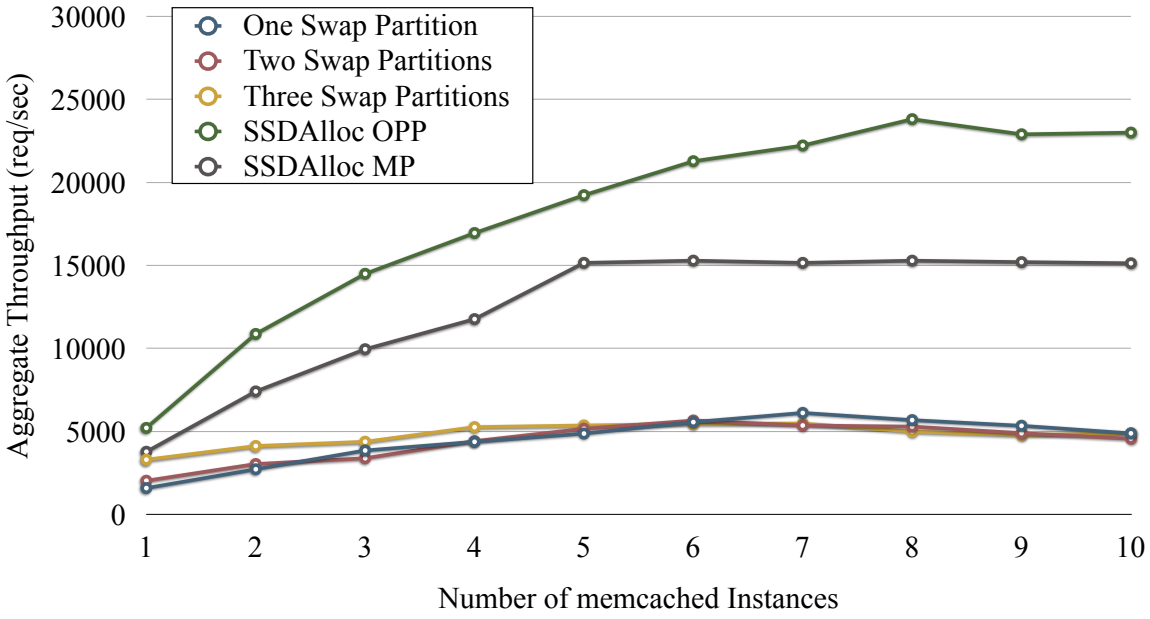

Figure 3.7: Memcache Results: OPP outperforms MP and SSD-swap by factors of 1.6 and 5.1 respectively (mix of 4byte to 4KB objects)

Figure 3.7 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, and memcached with one, two or 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% hit-rate), 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 has much higher variance in general. For SSD-Swap, the average response time is 667 microseconds and the standard deviation is 398 microseconds, as opposed to OPP's response times of 287 microseconds with a 112 microsecond standard deviation (high variance due to synchronous GC).

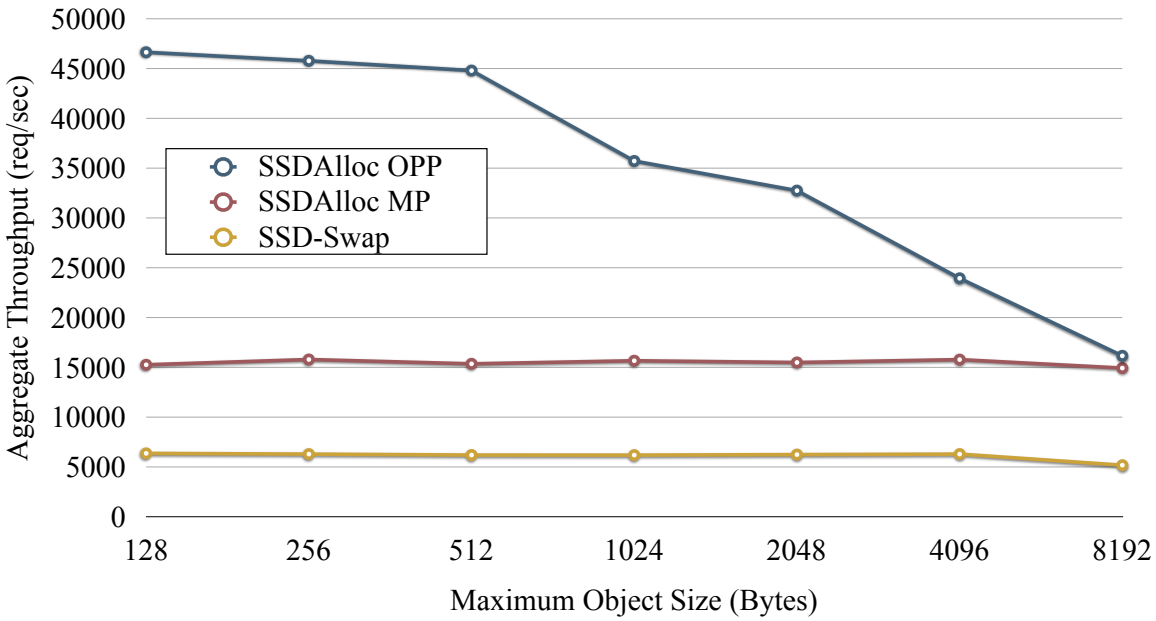

Figure 3.8: Memcache Results: SSDAlloc's use of objects internally can yield dramatic benefits, especially for smaller memcached objects

Figure 3.8 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 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 the case of MP and SSD-swap when moving from 4KB object size limit to 8KB is because the runtime 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.

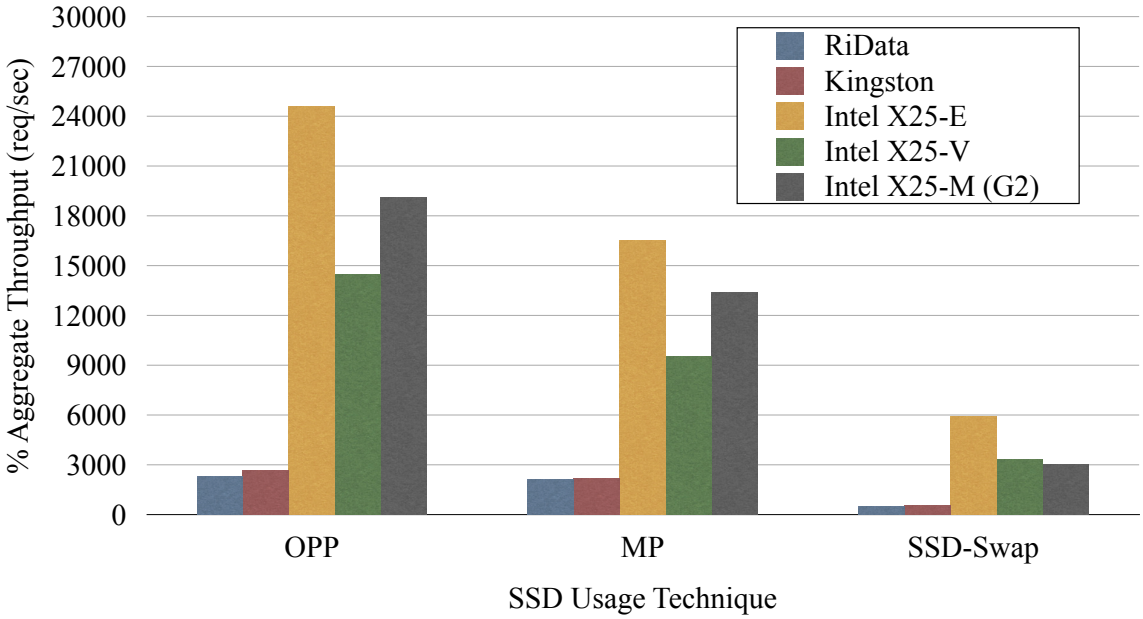

Figure 3.9: Memcache Results: SSDAlloc beats SSD-Swap by a factor of 4.1 to 6.4 for memcache tests (mix of 4byte to 4KB objects)

Figure 3.9 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 the object size distribution, OPP writes anywhere between **3.88–31.6** times more efficiently then 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.

3.4.3 Packet Cache Benchmarks

Packet caches (and chunk caches) built using SSDs scale the performance of network accelerators [4] and in-line data deduplicators [32] by exploiting the good random read performance and the large capacity of flash. Similar capacity DRAM-only systems will cost much more and also consume more power. We built a packet cache back-end 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. The 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 back-end. 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 configure 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.

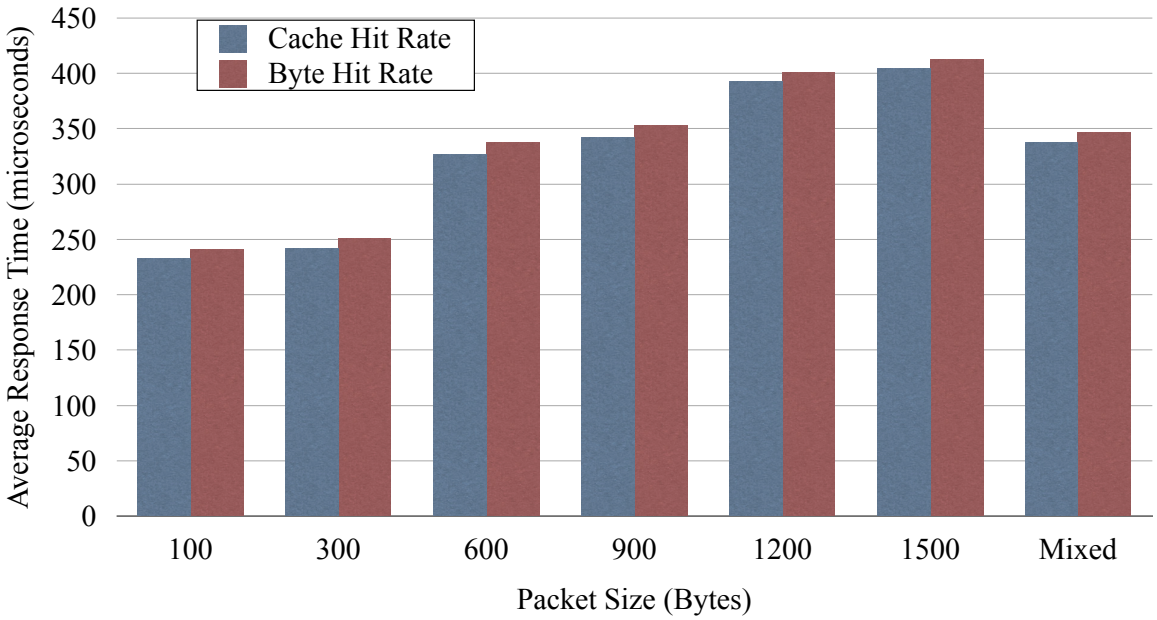

Figure 3.10: Packet Cache Benchmarks: SSDAlloc's runtime mechanism adds only up to 20 microseconds of latency overhead, while there is no significant difference in throughput

Figure 3.10 compares the response times of the OPP method using the transparent techniques described in this chapter 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, and there is no significant difference in throughput. The highest overhead observed is 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 performs **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. The write efficiency of OPP scales 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.

3.4.4 B+Tree Benchmarks

We built a B+Tree data structure via Boost framework [17] 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, 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*.

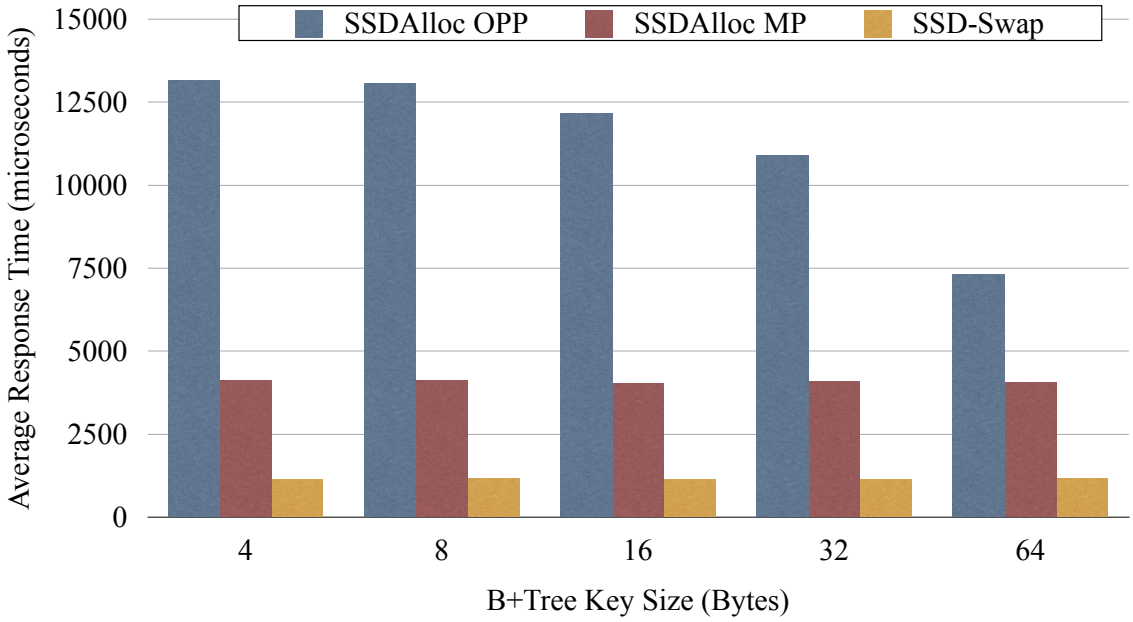

Figure 3.11: B+Tree Benchmarks: SSDAlloc's ability to internally use objects beats page-sized operations of MP or SSD-swap

Figure 3.11 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.

3.4.5 HashCache Benchmarks

Our final application benchmark is the efficient Web cache/WAN accelerator index based on HashCache [11]. 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 (up to 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 [54] 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).

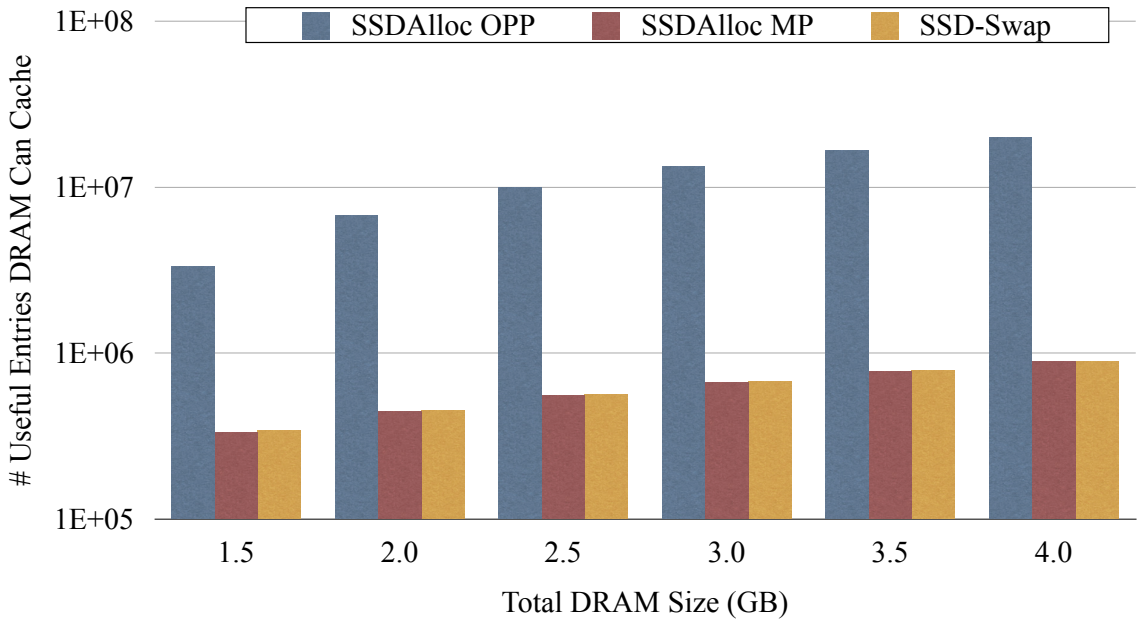

Figure 3.12: HashCache benchmarks: SSDAlloc OPP option can beat MP and SSD-Swap on RAM requirements due to caching objects instead of pages. The maximum size of a completely random working set of index entries each allocation method can cache in DRAM is shown (in log scale).

Figure 3.12 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 Object

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).

3.5 Summary

We introduce SSDAlloc, a hybrid main memory management system that allows developers to treat solid-state disk (SSD) as an extension of the DRAM in a system. SSDAlloc moves the SSD upward in the memory hierarchy, usable as a larger, slower form of DRAM instead of just a replacement/cache for the hard drive. By eliminating the usage of multiple layers of software part of the operating system and the filesystem, SSDAlloc enables a low-latency direct access of the SSD at the application level. Using SSDAlloc, applications can nearly transparently extend their memory footprints to hundreds of gigabytes and beyond without restructuring, well beyond the DRAM capacities of most servers. SSDAlloc presents an interface similar to that of `malloc` making it straightforward for developers to adopt and build new applications or port existing ones to use SSDs to augment DRAM in a system. Additionally, SSDAlloc can extract 90% of the SSDs 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 SSDs raw performance. SSDAlloc obtains these benefits

by transparently working at object granularity unlike traditional transparent systems that work at a page/block granularity.

Chapter 4

Conclusions and Future Work

Caches in various forms provide the necessary speedup for various Internet services including the Web, data delivery, and file synchronization. Caches help reduce the load on the network links and the servers by storing reusable and redundant content closer to the clients. Despite their importance, caches today are constrained by the trends in the evolution of today's memory (DRAM) and storage (disk) technologies. The current caches require a huge amount of DRAM for indexing large disks, increasing their cost. Furthermore, low disk speeds force the cache to use DRAM to serve content for high performance and further drive up the cost. New memory technologies like NAND-Flash offers much higher capacity than DRAM and a much higher performance than disks.

The trends in the evolution of DRAM, disk and new memory technologies like NAND-Flash warrant the rethinking of the design of caches to make effective use of each of these technologies. Towards these goals, this dissertation has attempted to answer the following questions:

1. How can one design a cache that can index disks that are larger by an order of magnitude than existing designs without using any additional memory?

2. What are the performance implications of such a cache design? More specifically, can such a design match the performance of existing cache designs?

3. How must new memory technologies like NAND-Flash be used to reduce the reliance of such caches on traditional memory and disk for performance?

4. How can one adopt such new memory technologies transparently, while masking their limitations? More specifically, can applications use them without any modifications?

We have addressed the first two questions in Chapter 2, and the final two questions in Chapter 3.

4.1 Rethinking cache indexing for memory efficiency

In Chapter 2 we developed HashCache, a high-performance configurable cache storage for 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.

4.2 Adopting new memory technologies

In Chapter 3, we developed SSDAlloc, which integrates new memory technologies like NAND-Flash into the virtual memory hierarchy of applications. Specifically, SSDAlloc provides a hybrid memory management system that allows new and existing applications to easily use NAND-Flash SSDs to augment the DRAM in a system. Such a system helps applications reduce their dependence on DRAM and disk for performance. Application state (like an index for a cache) can reside partly on NAND-Flash and reduce the DRAM requirements and thereby reduce the cost. Applications can also cache data from disk in NAND-Flash, reduce the pressure on the disk and thereby increase the performance.

SSDAlloc helps applications perform up to 17 times better than using the SSD as a swap space. Additionally, it helps applications perform up to 3.5 times better than using the SSD as a log-structured swap space. Furthermore, it can increase the SSD's lifetime by a factor of up to 30 times by transparently working at an object granularity as opposed to a page/block granularity. When the application modifies objects, only these changed objects need to be written to the SSD and not an entire page or a block containing the object. This leads to a drastic reduction in the amount of data written to the SSD for a given write workload.

SSDAlloc provides all the above benefits with minimal code changes. These changes are limited to the memory allocation part of the application code. In the applications that we used, we modified less than 0.05% of the original code to obtain the benefits of SSDAlloc. Additionally, the modifications allow the usage of the familiar memory allocation interface.

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 (Memcache), a B+Tree index, a packet cache back-end and an efficient hash table representation (HashCache), which required only minimal code changes, little application knowledge, and no expertise with the inner workings of SSDs.

4.3 Future Work

Reduce the page table overhead of SSDAlloc. The design of Object Tables as described in Chapter 3 attempts to minimize the overhead from the address translation structures inside SSDAlloc. However, each object created using OPP model requires a separate page table entry – an 8 byte value for 64 bit architectures. Furthermore, the Flash Translation Layer (FTL) inside the SSD performs its own address

translation from the logical block address at the OS level to the physical block address at the device level. Modern FTLs use host DRAM for storing these address translations [46].

In the future, it would be beneficial to reduce these multiple levels of address translations and perform all the necessary address translation at one place to reduce latency and save DRAM space. Since we propose that NAND-Flash be used via the virtual memory, it is natural to propose that this address translation be performed by overloading the page tables of the application's process itself.

Such a method of address translation presents many benefits. Firstly, traditional OS's page tables are designed so that they can be paged in and out of persistent storage in an on-demand fashion, thereby reducing the memory pressure inside the OS. Secondly, virtual memory to physical memory address conversion is a fairly well understood and optimized sub-system of an OS which would make adoption of NAND-Flash that much easier. Finally, the hierarchical model of the page table address conversions not only provides low-latency conversions but also takes into account spatial and temporal localities in the workload.

Reduce the page fault overhead in SSDAlloc. The biggest overhead in SSDAlloc is the latency due to the page fault generation. There are multiple entities that contribute to this latency and each of these entities creates an opportunity to optimize existing operating systems to make them more accepting of new memory technologies. The biggest source of latency is the serialization of multiple threads that modify the page tables at the same time. Recent work has suggested the usage of message passing, "lock-free" data structures and fine grained locking for managing the virtual memory mappings of a process to address this problem for soft faults [14, 18, 25]. However, SSDAlloc primarily deals with hard faults and has many more data structures that are shared across threads, e.g., the garbage collector and would require a more thorough study before one can address the problem.

Rethink paging over the network. Optical networks within the data center [41, 89, 108] are fast narrowing the gap between a local device and a networked device. This development can help servers within a cluster to fetch data not only from the local SSD at a low latency but also from a remote SSD. The high latency of NAND-Flash could also mask the network latency inside an optical network. Such a low-latency network will warrant the reopening of shared-memory clusters. However, the bottleneck this time would not be the network, but it would be the overhead from coordination of memory within the several systems in the cluster.

Recent work has suggested the usage of specialized hardware-based clusters to solve the problem of coordination [13]. Modern NAND-Flash devices use powerful processors to perform FTL operations on the flash device itself. In the future, the additional processing power on these devices can themselves be used for coordinating memory accesses between these various processors within the cluster and further reduce the complexity of coordination of devices within a low-latency network.

Adopting low latency solid state memory technologies. While NAND-Flash's latency is four orders of magnitude higher than that of DRAM [20], other new memory technologies like PCM have much lower latency – only an order of magnitude more than DRAM [63]. The current method of servicing page faults at the user-space after a context switch would therefore be a poor model for a PCM-based SSD. In the future, a safe yet low-overhead way of servicing hard page faults would be desirable to incorporate such low-latency solid state memory technologies into the virtual memory hierarchy.

While SSDs (NAND-Flash or PCM based) are not byte-addressable, other new memory technologies like JEDEC-based PCM modules are [85]. Also, they have a high enough write endurance to allow more frequent in-place writes. This creates an interesting optimization problem with respect to what physical memory technology to use for a given virtual memory page. Scalable monitoring techniques have to be put

in place to help virtual memory pages to use the right physical memory technology. While such techniques have been explored for the storage sub-system [27], no work that addresses this problem in the memory sub-system exists. An example scenario where such a technique would be useful is the following: one must switch from PCM-based physical pages to DRAM-based one when the virtual memory address involved is being written often. However, the latency of PCM is low-enough to discourage such changes at a regular interval. The page faults in such cases where a virtual memory address migrates from a PCM page to a DRAM one would be a threshold-based one rather than an on-demand one.

Bridging the gap between the application and the devices. A modern enterprise storage system is likely to contain: DRAM used as a page cache, a NAND-flash-based block cache, and finally, a SAN composed of its own DRAM/Flash caching layers in front of an array of disk drives. Today, neither the OS nor the application has any control over how data is managed within these tiers. We propose a general purpose extension to the OS block layer that enables applications to express usage intent to generic block devices enabling performance optimization [7]. In the future, we wish to provide a convenient interface to these complex storage devices for the application to provide advises, issue directives, query for quality-of-service guarantees and perform custom tiering of data.

Bibliography

[1] ACM Digital Library of Articles Published by the ACM.
`http://dl.acm.org/`.

[2] Nitin Agarwal, Vijayan Prabhakaran, Tedd Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. USENIX ATC*, Boston, MA, June 2008.

[3] Akamai Technologies Inc.
`http://www.akamai.com/`.

[4] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *Proc. 7th USENIX NSDI*, San Jose, CA, April 2010.

[5] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proc. 22nd ACM SOSP*, Big Sky, MT, October 2009.

[6] arXiv: E-Print Service for Basic Sciences.
`http://arxiv.org/help/general`.

[7] Anirudh Badam and David W. Nellans. Enabling Application Directed Storage Devices. In *Proc. 3rd NVMW*, San Diego, CA, March 2012.

[8] Anirudh Badam and Vivek S. Pai. Beating Netbooks into Servers: Making Some Computers More Equal Than Others. In *Proc. 3rd ACM NSDR*, BigSky, MO, October 2009.

[9] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid RAM/SSD Allocation Made Easy. In *Proc. 2nd NVMW*, San Diego, CA, March 2011.

[10] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proc. 8th USENIX NSDI*, Boston, MA, March 2011.

[11] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proc. 6th USENIX NSDI*, Boston, MA, March 2009.

[12] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Proc. 5th ACM ASPLOS*, Boston, MA, October 1992.

[13] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proc. 9th USENIX NSDI*, San Jose, CA, April 2012.

[14] Andrew Baumann, Simon Peter, Adrian Schupbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your Computer is Already a Distributed System. Why isn't your OS? In *Proc. 11th USENIX HotOS*, Monte Verita, Switzerland, May 2009.

[15] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A Design for High-Performance Flash Disks. *SIGOPS OSR*, 41(2):88–93, April 2007.

[16] Burton H. Bloom. Space/Time Trade-offs in Hash Coding With Allowable Errors. *CACM*, 13(7):422–426, July 1970.

[17] Boost Template Library.
`http://www.boost.org/`.

[18] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010.

[19] Eric Brewer, Paul Gauthier, and Dennis McEvoy. Long-term viability of large-scale caches. In *Proc. 3rd WWW Caching Workshop*, Manchester, England, June 1998.

[20] Jeffery A. Brown and Dean M. Tullsen. The Shared-Thread Multiprocessor. In *Proc. 22nd ACM ICS*, Island of Kos, Greece, June 2008.

[21] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. Hac: Hybrid adaptive caching for distributed storage systems. In *Proc. 16th ACM SOSP*, Saint-Malô, France, October 1997.

[22] CERN Cache.
`http://www.w3.org/Daemon/`.

[23] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. In *Proc. USENIX ATC*, San Diego, CA, January 1996.

[24] Citrix Systems.
`http://www.citrix.com/`.

[25] Austin Clements, M. Franz Kaashoek, and Nickolai Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *Proc. ACM ASPLOS*, London, United Kingdom, March 2012.

[26] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe With Next-Generation, Non-Volatile Memories. In *Proc. ACM ASPLOS*, Newport Beach, CA, March 2011.

[27] Jeremy Condit, Edmund B. Nightingale, Cristopher Frost, Engin Ipek, Doug Burger, Benjamin Lee, and Derrick Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proc. 22nd ACM SOSP*, Big Sky, MT, October 2009.

[28] Alan L. Cox, Y. Charlie Hu, Vijay S. Pai, Vivek S. Pai, and Willy Zwaenepoel. Storage and retrieval system for WEB cache. U.S. Patent 7231494, 2000.

[29] CPU Timing Calibrator. `http://homepages.cwi.nl/~manegold/Calibrator/`.

[30] Datacenter Map.
`http://www.datacentermap.com/`.

[31] Datadomain.
`http://www.datadomain.com/`.

[32] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proc. USENIX ATC*, Boston, MA, June 2010.

[33] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. In *Proc. 36th VLDB*, Singapore, Singapore, September 2010.

[34] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash. In *Proc 30th ACM SIGMOD*, Athens, Greece, June 2011.

[35] Doctors Without Borders.
`http://www.doctorswithoutborders.org/aboutus/?ref=main-menu`.

[36] Bowei Du, Michael Demmer, and Eric Brewer. Analysis of WWW traffic in Cambodia and Ghana. In *Proc. 15th WWW*, Edinburgh, Scotland, May 2006.

[37] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. Primary Data DeduplicationLarge Scale Study and System Design. In *Proc. USENIX ATC*, Boston, MA, June 2012.

[38] EMC VFCache: Server Flash Cache.
`http://www.emc.com/storage/vfcache/vfcache.htm`.

[39] Facebook: An Online Social Network.
https://www.facebook.com/facebook?sk=info.

[40] Facebook Memcache Requirements.
https://www.facebook.com/note.php?note_id=39391378919.

[41] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proc. ACM SIGCOMM*, New Delhi, India, August 2010.

[42] Anja Feldmann, Ramon Caceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *Proc. 18th IEEE INFOCOM*, New York, NY, March 1999.

[43] Michael J. Freedman, Eric Freudenthal, and David Mazieres. Democratizing content publication with coral. In *Proc. 1st USENIX NSDI*, San Francisco, CA, March 2004.

[44] Fusion-io Directcache: Transparent Storage Accelerator.
http://www.fusionio.com/systems/directcache/.

[45] Fusion-io: ioDrive Octal.
http://www.fusionio.com/platforms/iodrive-octal/.

[46] Fusion-io's FTL: Memory Overhead.
http://www.tomshardware.com/reviews/fusion-io-ioxtreme-ssd,
2488-3.html.

[47] Syam Gadde, Jeff Chase, and Michael Rabinovich. A Taste of Crispy Squid. In *Proc. 1st Workshop on Internet Server Performance*, Madison, WI, June 1998.

[48] Gallup: An Online Polling Service.
http://www.gallup.com/corporate/115/About-Gallup.aspx.

[49] Google News: An Online News Aggregator.
http://news.google.com/intl/en_us/about_google_news.html.

[50] Steven Gribble and Eric Brewer. System design issues for internet middleware services: Deductions from a large client trace. In *Proc. 1st USITS*, Monterey, CA, December 1997.

[51] Martin Grund, Jens Krueger, Hasso Plattner, Alexander Zeier, and Philippe Cudre-Mauroux Samual Madden. HYRISE–A Main Memory Hybrid Storage Engine. In *Proc. 36th VLDB*, Singapore, Singapore, September 2010.

[52] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proc. ACM ASPLOS*, Washington, DC, March 2009.

[53] Sunghwan Ihm and Vivek S. Pai. Towards Understanding Modern Web Traffic. In *Proc. ACM IMC*, Berlin, Germany, November 2011.

[54] Sunghwan Ihm, KyoungSoo Park, and Vivek S. Pai. Wide-area Network Acceleration for the Developing Regions. In *Proc. USENIX ATC*, Boston, MA, June 2010.

[55] InnoDB: MySQL Transaction Engine. `http://www.innodb.com/`.

[56] Intel Classmate PC. `http://www.intel.com/content/www/us/en/intel-learning-series/technology-to-classroom.html`.

[57] Internet World Statistics - Cost of the Internet in the USA. `http://www.internetworldstats.com/am/us.htm`.

[58] Internet World Statistics - World Penetration. `http://www.internetworldstats.com/stats.htm`.

[59] iTunes: An Online Educational Resource. `http://www.apple.com/education/itunes-u/`.

[60] Shudong Jin and Azer Bestavros. Popularity-Aware GreedyDual-Size Web Proxy Caching Algorithms. In *Proc. 20th ICDCS*, Taipei, Taiwan, April 2000.

[61] Taeho Kgil and Trevor N. Mudge. Flashcache: A NAND Flash Memory File Cache for Low Power Web Servers. In *Proc. CASES*, Seoul, Korea, October 2006.

[62] Sohyang Ko, Seonsoo Jun, Yeonseung Ryu, Ohhoon Kwon, and Kern Koh. A New Linux Swap System for Flash Memory Storage Devices. In *Proc. ICCSA*, Perugia, Italy, June 2008.

[63] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proc. 36th ACM SIGARCH*, New York, NY, June 2009.

[64] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proc. 27th ACM SIGMOD*, Vancouver, Canada, June 2008.

[65] Peter Van Den Linden. Expert C Programming: Deep C Secrets. Prentice Hall, 1994.

[66] Carlos Maltzahn, Kathy Richardson, and Dirk Grunwald. Reducing the disk I/O of Web proxy server caches. In *Proc. USENIX ATC*, Monterey, CA, June 1999.

[67] Evangelos P. Markatos, Dionisios N. Pnevmatikatos, Michail D. Flouris, and Manolis G.H. Katevenis. Web-Conscious Storage Management for Web Proxies. *IEEE/ACM Trans. on Networking*, 10(6):735–748, December 2002.

[68] Memcache.
`http://www.danga.com/memcached`.

[69] MIT Open Course Ware.
`http://ocw.mit.edu/about/`.

[70] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. on Parallel and Distributed Systems*, 12(10):1094–1104, October 2001.

[71] Jeff 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.

[72] Jeffrey C. Mogul, Yee Man Chan, and Terence Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proc. 1st USENIX NSDI*, San Francisco, CA, March 2004.

[73] Dushyanth Narayanan, Eno Thereska, Austin Donelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds, analysis of tradeoffs. In *Proc. EUROSYS*, Nuremberg, Germany, March 2009.

[74] Netflix: An Online Video Streaming Service.
`https://signup.netflix.com/MediaCenter/HowNetflixWorks`.

[75] Netflix CDN Caching Server Configuration.
`https://signup.netflix.com/openconnect/hardware`.

[76] O3b Networks.
`http://www.o3bnetworks.com/`.

[77] OLPC.
`http://www.laptop.org/`.

[78] OLPC Laptop Configuration.
`http://wiki.laptop.org/go/Hardware_specification`.

[79] OLPC Server Configuration.
`http://wiki.laptop.org/go/XS_Recommended_Hardware`.

[80] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS OSR*, 43(4):92–105, January 2010.

[81] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proc. USENIX ATC*, Monterey, CA, June 1999.

[82] KyoungSoo Park and Vivek S. Pai. Scale and Performance in the CoBlitz Large-file Distribution Service. In *Proc. 3rd USENIX NSDI*, San Jose, CA, May 2006.

[83] Rabin Patra, Sergiu Nedevschi, Sonesh Surana, Anmol Sheth, Lakshminarayanan Subramanian, and Eric Brewer. WiLDNet: Design and implementation of high performance wifi based long distance networks. In *Proc. 4th USENIX NSDI*, Cambridge, MA, April 2007.

[84] PT Malloc Memory Manager.
`http://www.malloc.de/en/`.

[85] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Viji Srinivasan, Luis Lastras, and Bulent Abali. Enhancing Lifetime and Security of Phase Change Memories via Start-Gap Wear Leveling. In *Proc. 42nd IEEE MICRO*, New York, NY, December 2009.

[86] Sean Rhea, Kevin Liang, and Eric Brewer. Value-based web caching. In *Proc. 12th WWW*, Budapest, Hungary, May 2003.

[87] Riverbed Technology, Inc.
`http://www.riverbed.com/`.

[88] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. on Computer Systems*, 10(1):26–52, February 1992.

[89] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. Its Time for Low Latency. In *Proc. 13th USENIX HotOS*, Napa, CA, May 2011.

[90] Mark Russell and Tim Hopkins. CFTP: A Caching FTP Server. *Computer Networks and ISDN Systems*, 30(22–23):2211–2222, November 1998.

[91] Mohit Saxena and Michael Swift. FlashVM: Virtual Memory Management on Flash. In *Proc. USENIX ATC*, Boston, MA, June 2010.

[92] André Seznec. A Case or Two-Way Skewed-Associative Caches. In *Proc. 20th ISCA*, San Diego, CA, May 1993.

[93] Elizabeth A. M. Shriver, Eran Gabber, Lan Huang, and Christopher A. Stein. Storage Management for Web Proxies. In *Proc. USENIX ATC*, Boston, MA, June 2001.

[94] Silver Peak Systems, Inc.
`http://www.silver-peak.com/`.

[95] Skype: An Online Audio/Video Calling Service.
http://about.skype.com/.

[96] Squid.
http://www.squid-cache.org/.

[97] Squid Memory Usage.
http://www.comfsm.fm/computing/squid/FAQ-8.html.

[98] Lakshminarayan Subramanian, Sonesh Surana, Rabin Patra, Sergiu Nedevschi, Melissa Ho, Eric Brewer, and Anmol Sheth. Rethinking Wireless in the Developing World. In *Proc. 5th USENIX HotNets*, Irvine, CA, November 2006.

[99] Kshitij Sudan, Anirudh Badam, and David W. Nellans. NAND-Flash: Fast Disk or Slow Memory? In *Proc. 3rd NVMW*, San Diego, CA, March 2012.

[100] The Measurement Factory - Fourth Cacheoff Results.
http://www.measurement-factory.com/results/public/cacheoff/N04/report.by-alph.html.

[101] The Measurement Factory - Polymix Cache Workload.
\http://www.web-polygraph.org/docs/workloads/polymix-4/.

[102] The Measurement Factory - Third Cacheoff Results.
http://www.measurement-factory.com/results/public/cacheoff/N03/report.by-alph.html.

[103] The Measurement Factory - Web Polygraph.
\http://www.web-polygraph.org/.

[104] TPCC Benchmark.
http://www.tpc.org/tpcc/.

[105] Twitter: A Real Time Information Network.
https://twitter.com/about.

[106] Verivue Inc.
http://www.verivue.com/.

[107] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proc. ACM ASPLOS*, Newport Beach, CA, March 2011.

[108] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T. S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-through: Part-time optics in data centers. In *Proc. ACM SIGCOMM*, New Delhi, India, August 2010.

[109] Limin Wang, KyoungSoo Park, Ruoming Pang, Vivek Pai, and Larry Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Proc. USENIX ATC*, Boston, MA, June 2004.

[110] WiMax Forum.
http://www.wimaxforum.org/home/.

[111] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An efficient b-tree layer for flash-memory storage systems. In *Proc. 10th RTCSA*, Gothenburg, Sweden, August 2004.

[112] Michael Wu and Willy Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proc. 6th ACM ASPLOS*, San Jose, CA, October 1994.

[113] Youtube: An Online Video Streaming Service.
http://www.youtube.com/t/about_youtube.