NEW INTERFACES FOR SOLID-STATE MEMORY MANAGEMENT

by

Mohit Saxena

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

Date of final oral examination: 05/09/2013

The dissertation is approved by the following members of the Final Oral Committee:

Michael M. Swift, Assistant Professor, Computer Sciences

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences

Remzi H. Arpaci-Dusseau, Professor, Electrical and Computer Engineering

Mark D. Hill, Professor, Computer Sciences

Arif Merchant, Research Scientist, Google

To my parents, brother and sister-in-law.

ACKNOWLEDGMENTS

Thanks everyone! I would like to begin by expressing my deepest gratitude to the most wonderful teacher in my life, Mike Swift. The journey started when I first met him on the Welcome Weekend for prospective graduate students in March 2008. In about half an hour of discussion with him, it was decided this is it, and since then it has been a great pleasure for me working under his guidance. He taught me how to do good and practical research, instilled into me the virtues of patience and persistence, and greatly helped me groom up professionally. I learned from him how to think deep yet with the big picture in mind, how to challenge my own ideas to bring out the best from them, and most importantly the value of a good story to present my work. He has been my role model.

I would also like to thank the Wisconsin faculty with whom I closely interacted with: Remzi Arpaci-Dusseau and Mark Hill. Remzi has been a great source of inspiration to me all through my Ph.D. His precise comments on my work, paper drafts, and prelim proposal made me revisit my research style and problems time and again. I have learned from him directly through discussions with him, and indirectly through interaction with his several students. I am also grateful to Mark for serving on my committee. I learned from him the value of summarizing ideas comprehensively and never losing focus of long-term vision for research impact. I also thank him for encouraging me to pursue a difficult endeavor of working with real hardware towards the end of my Ph.D. It has been a very satisfying and insightful experience for me.

I would also like to thank Andrea Arpaci-Dusseau and Arif Merchant for serving on my committee. Andrea has always been a great source of advice, and I very much enjoyed taking her Distributed Systems class in my first year. Arif has always been a great and very supportive mentor during my internship at HP Labs and afterwards during my job search. This internship opportunity offered me an early exposure to learn about the nuances of corporate research, which I would certainly require as I will be moving to a corporate research lab in the near future. I also thank Mehul Shah and Stavros Harizopoulos for their great guidance and sharing their expertise with me on database transactions and reliability

during my internship.

I would also like to extend my sincerest thanks to faculty members at Purdue: Ramana Kompella and Sonia Fahmy. I thank both of them for instilling into me the passion for pursuing Ph.D. by being my first mentors in graduate school, and also encouraging me to attend conferences during my M.S. at Purdue.

I would also like to thank all graduate students with whom I worked on different projects. I enjoyed working with Yiying Zhang on two projects and the wonderful collaboration has resulted in two published papers. Both of us learned together several sweet and bitter lessons working with disk simulators, emulators and hardware prototypes over the last three years. I also thank Umang Sharan and Puneet Gupta for working with me on two projects at Purdue and IIT-Delhi.

I would also like to thank all the Wisconsin systems students that I had the pleasure to interact during the years, including Nitin Agrawal, Ishani Ahuja, Ashok Anand, Leo Arulraj, Arkaprava Basu, Theophilus Benson, Vijay Chidambaram, Thanh Do, Chris Dragga, Tyler Harter, Asim Kadav, Lanyue Lu, Ashish Patro, Thanumalayan Pillai, Sankaralingam Panneerselvam, Abhishek Rajimwale, Matt Renzelmann, Thawan Kooburat, Srinath Sridharan, Sriram Subramanian, Swaminathan Sundararaman, Andres Jaan Tack, Venkatanathan Varadarjan, Haris Volos, Zev Weiss, Wei Zhang, and Yupu Zhang. I also thank them for attending the weekly OS reading group meetings, and for the insightful discussions we engaged into for understanding research sometimes well beyond our areas of research.

I feel lucky to have Andres, Sankar, Thawan and Venkat as great office-mates over the last five years. I also thank all my research siblings – Haris, Matt, Asim, Arka, Sankar and Venkat – for the valuable help and advice they offered me during lunch, coffee breaks and practice talks. I also enjoyed the several research discussions I had with Vijay on file systems, Arka on processor caches, and Abhishek on flash management in my first year.

I would like to especially thank Tushar Khot and Varghese Matthew at Wisconsin, and Abhinav Pathak, Harinath Reddy and Umang Sharan at Purdue, for being some of the best friends that I will treasure for my life. I really enjoyed the several hiking trips with Varghese,

and greatly thank him for his selfless help and car rides all through these years. I will miss the dinners, movies, badminton and cricket matches with Tushar. I feel happy to re-unite with Abhinav and Umang again in California after five years.

I would also like to thank Florentina Popovici at Google Madison, Lakshmi Bairavasundaram and James Lentini at NetApp ATG, and George Candea at EPFL for inviting me to give talks on my research. I also sincerely acknowledge the Lawrence Landweber Fellowship, various NSF Grants, and a NetApp Gift for supporting my research all through these years.

Finally, I would like to thank my wonderful family, without whose love and support this Ph.D. would have not been possible. I am extremely grateful to my parents who have always supported me with selfless love and strong faith in my abilities. To the most important person in my life, my elder brother, a big thanks! There are no words to express my gratitude for his unconditional love and valuable advice all through my life. I would also like to thank my sister-in-law for her support and encouragement, and tasty home cooked food in US over the last two years. I dedicate this thesis to my family.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The availability of high-speed flash solid-state devices (SSD) have introduced a new tier into the memory hierarchy. SSDs have dramatically different properties than disks, yet are exposed in many systems as a generic block device. This dissertation presents the design, implementation, and evaluation of two systems that update the interface to these devices to better match their capabilities as a new memory tier.

The first part of this dissertation is a new flash virtual memory system called FlashVM, which extends main memory by virtualizing it with inexpensive flash storage. FlashVM revisits the various paging mechanisms in the core virtual memory subsystem of the Linux kernel, which have been optimized for the characteristics of disks. It de-diskifies these mechanisms by modifying the paging system along code paths for allocating, reading and writing back pages to improve performance and reliability with flash SSDs. Overall, we find that our optimizations in Linux for paging to dedicated flash improve performance by up to 95% and reduce the number of flash writes by up to 93% compared to Linux VM subsystem.

The second system called FlashTier focuses on the use of flash SSDs as a cache in front of slower disks. FlashTier identifies the various differences between the interface offered by an SSD, a persistent block store, and the service it provides, caching data. It redresses these differences with a new solid-state cache (SSC) device, which exposes a new cache-aware device interface to distinguish between clean and dirty state of cached data blocks. The SSC also has a new block-addressing mechanism, which unifies address translation maps across host OS and device. Finally, the SSC leverages the block state of cached data to provide a new flash space management technique. We demonstrate the benefits of SSC design in simulation as well as by prototyping it on the OpenSSD hardware platform. FlashTier improves cache performance by up to 168% performance improvement over consumer-grade SSDs and up to 52% over high-end SSDs. It also improves flash lifetime for write-intensive workloads by up to 60% compared to SSD caches with a traditional block interface.

1 INTRODUCTION

This dissertation presents new approaches to integrate non-volatile memory (NVM) technologies in the memory hierarchy to improve the performance and reliability of computer systems. We present the design, implementation and evaluation of new operating system abstractions and hardware interfaces for using these technologies as memory or disk extension. We target the most commercially ubiquitous NVM technology — flash solid-state drives (SSDs) — and implemented mechanisms that provide better software support for OS and applications to manage SSDs as a new memory tier.

1.1 Motivation

Three factors motivated this thesis: (1) advent of new memory technologies, (2) lack of software abstractions that best match their price and performance, and (3) inadequate interfaces between hardware and OS or applications to manage them.

1.1.1 Technology Trends

Old Memory Hierarchy. Over the last four to five decades, there have been mostly two levels in the memory hierarchy.

Memory or DRAM can provide millions to billions of operations per second. However, it is volatile and slow to fill from durable secondary medium. Further, it has not scaled up in terms of capacity and price as fast as disks over the last one decade. Table 2.1 (see Section 2.1) lists the spot-price of DRAM, about USD $5 per gigabyte. In addition, the high power consumption and super-linear price increase for each additional DIMM slot in a system limits the amount of data that can be stored in memory. For example, it takes up to 14.4 K Watts to power and cool 4 TB of DRAM and about 80 rack units of space for a SPARC M9000 Server [83].

In contrast, disks have advanced in terms of capacity and areal density at a pace of about 40% per year to sustain the enormous growth in data sets over the last one decade [54]. Today,

a consumer-grade 7200 RPM disk costs no more than 10 cents per gigabyte (see Table 2.1). By 2020, it is projected that disks would achieve an areal density of 10 TB/in^2, which would enable a single 2.5 inch disk to store over 7 TB of data at a cost of USD \$3/TB [54]. However, disks have barely managed to scale well in terms of performance: operations per second and access latencies. Even the fastest disks can not provide more than a few hundred random operations per second with peak latencies as high as 7 ms. Sequential bandwidth for disks has improved with disk technology and is of the order of 200-300 MB/s.

As a result, it has been easy to find the right tier to store data: whatever is small and requires high performance resides in memory, and large datasets with durability requirements are stored on disk. In addition, sequentially referenced data also benefits from the high bandwidth of disks.

Past: Memory Technologies. To bridge the price and performance gap between memory and disk, there have been several past attempts to use alternate memory technologies as an intermediate tier in the memory hierarchy. Most such attempts have come in the context of non-volatile memory technologies, for example, bubble memory [18] in the late 1970s, and MEMS [112] in the early 2000s. Bubble memory never scaled over 1 Mb per chip and access latencies were longer than 1 ms. MEMS were better off with 500 MB per chip and latencies of the order of a few microseconds. However, none of these technologies made their way into commercial products, primarily because they never got as performant as DRAM or cheap enough to be used in large capacities to cache disk data.

Present and Future: Flash. Flash memory [42] is the first ubiquitously available memory technology after memory and disk. Commonly used NAND flash is non-volatile and block addressable. It first penetrated into consumer products through mobile systems, for example, smartphones, tablets and laptops because of its small form factor and low power requirements. About 21 exabytes was fabricated world-wide in 2011, and it reached *36 exabytes* in 2012. In comparison, disk manufacturers shipped close to 100 exabytes of disk storage into the laptop market in 2011 alone [101]. The majority of flash still comes embedded in mobile devices, smartphones, tablets and memory cards. Only a small fraction, about 12% of fabricated flash,

is packaged as solid-state disks (SSD), which is about 23 million SSDs, less than 9% of the laptop disk units shipped in 2011 [101].

SSDs have scaled to terabytes of capacity recently with performance intermediate to memory and disk (see Table 2.1). They can provide read/write access to stored data in less than 200 µs, and are priced at USD $1 per gigabyte.

1.1.2 OS Abstractions

The price and performance of SSDs motivate their use as a disk replacement for storage or as an extension to memory. File system and virtual memory have been the two traditional approaches for the OS to abstract SSDs as storage or memory.

Flash: Memory or Storage. As a disk replacement, SSDs can be used as the primary medium for storage underneath a file system. In addition to faster file access, their small form factors and low power requirements has motivated their use as a storage device in mobile devices and laptops. Apple is one of the first companies to use SSDs for storage in Macbook Air. Google also ships Chromebooks with SSDs for primary storage.

However, SSDs are still an order of magnitude more expensive than disks (see Table 2.1), and therefore, not all datasets can be stored on just the SSDs in general purpose environments. Cold data with less value to the application can be stored more cheaply on disks with little loss in application performance.

Memory and Disk Extensions. As a memory extension, SSDs can be used to either replace some amount of DRAM or augment it with cheaper flash through virtual memory paging; or persist DRAM with durable flash.

Extending memory with cheaper flash is useful in mobile systems, laptops and desktops; which are usually constrained by the number of DIMM slots, price of memory that can be configured, and DRAM power requirements. In this thesis, we investigate how to extend memory through operating system paging to flash SSDs. We extend the virtual memory subsytem in the Linux kernel to adapt it to the unique performance and reliability properties of flash SSDs used as a swap device.

As a disk extension, SSDs can be used to provide higher performance for access to data stored on disk. SSDs enable fast random access and lower access latencies, which can be used to selectively cache the working sets of I/O workloads.

SSDs as a front-end cache to disks is useful in several environments: client or server caches with direct-attached SSDs, caches for network storage controllers, and caches for distributed file sytems or databases. In this thesis, we investigate a popular use of direct-attached SSDs as a cache to local disks. A cache manager in the OS at the block layer transparently interposes on file system I/O requests to underlying disk, and serves them from the SSD cache on a cache hit, or from the disk on a cache miss. This approach has several advantages, for example, it enables the use of commodity SSDs as a block cache and is completely transparent to the file system and application. However, we also identify inefficiencies with this approach arising from the differences between the interface offered by a commodity SSD, and the service it provides, caching data.

1.1.3 Existing Interfaces

The use of SSDs as an intermediate tier between memory and disk as virtual memory or block cache faces the challenge of retrofitting the same OS and application interfaces designed originally for memory or storage devices.

Memory and Storage Interfaces. Using memory interfaces such as *mmap* and *msync* for flash-backed extended memory abstraction is insufficient. For example, a few bytes of updates to memory require at least a full 4 KB dirty page to be flushed back to flash as part of the *msync* operation. In contrast, extended memory requires a more fine-grained commit operation than *msync* for well-defined consistency and durability semantics of memory.

Similarly, storage interfaces such as *read*, *write* and *fsync* have been designed for file systems and can not express about cache semantics. For example, caches have clean and dirty data, and there is no way we can communicate this information associated with a block from the OS or the application to the SSD for better flash management and performance.

In this thesis, we investigate ways in which small yet clever extensions to the existing

interfaces can suite the requirement of the new memory tier served by SSDs.

Performance and Features. The second problem with the use of existing SSD interfaces is the inability of the OS or applications to tap the best performance by exploiting features implemented within these devices. For example, almost all modern SSDs implement address translation and error correction for better performance and reliability. However, OS abstractions such as cache manager and file systems duplicate both these functionalities by maintaining another level of address maps and checksums.

In this thesis, we investigate how we can reduce performance and space overheads incurred due to duplication by leveraging the functionality implemented within the SSD through small extensions to the block interface without sacrificing any generality of our design.

1.1.4 Problem Statement

To summarize, this thesis addresses the following key question: *How can the OS and application software better manage flash SSDs as a new data tier?*

We design, implement, prototype and evaluate two different systems to address this question: using SSDs as a memory extension with FlashVM [97, 98], and using SSDs as a disk extension in FlashTier [99, 100]. We implement the FlashTier design twice, first in simulation and later on a real hardware platform.

Managing commodity SSDs as a new data tier using existing OS abstractions and hardware block interface raises three major problems. First, existing OS abstractions such as virtual memory paging have been optimized for disks and hurt performance with the use of SSDs. Second, commodity SSDs lack any coordination with software mechanisms implemented within the OS, for example address translation duplicated within SSDs and OS increases overheads for memory and crash consistency. Finally, the traditional block interface of SSDs has been designed for permanent data storage and loses opportunities for customization as a new data tier to achieve better performance and reliability for flash wear-out.

In the context of FlashVM, we find that optimizing the OS virtual memory subsystem mechanisms for the unique characteristics of SSDs is sufficient for improved performance and

reliability. However, we have to revisit the different paging mechanisms, sometimes de-diskify them from the assumptions inherent to disk properties, and in many cases adapt them to the unique properties of flash.

In the context of FlashTier, we find that in addition to optimizing existing OS mechanisms, it is imperative to customize the block interface to SSDs for caching. The custom interface allows the OS to exploit the internal device capabilities without duplicating its functionality. In addition, it improves performance and reliability by enabling more coordination between the OS and SSDs for flash and cache management.

1.2 Thesis and Contributions

The contributions of this thesis are improved performance, reliability and consistency for two state-of-the-art systems: the Linux virtual memory subsystem, and the Facebook FlashCache cache manager using commodity SSDs for caching deployed widely in production [28].

1.2.1 FlashVM: Large and Cheap Extended Memory

The contributions of FlashVM are: (1) an experimental analysis of the cost and benefits of dedicating flash for virtual memory, (2) optimizing the core virtual memory subsystem of the Linux kernel for the unique performance and reliability characteristics of SSDs, and (3) de-diskifying the mechanisms built to optimize for the properties of disks.

For our analysis, we use five different workloads from various computing domains including image processing, model checking, transaction processing, and object caching. We find that application execution time can be reduced by up to 94% compared to disk-backed paging at a small price increase of dedicating flash for paging. Alternatively, we find that by dedicating flash we can replace up to 84% of the memory configured in a system with performance identical to disk-backed paging resulting in both cost and power savings.

Second, we identify several paging mechanisms along the read, write and allocation code paths of the Linux virtual memory subsystem, which have been optimized for the performance characterisitics of disks over the last two decades. We implement 8 different new techniques

to improve performance and 3 techniques to reduce the number of writes for flash wear-out. We find that performance can be improved by 14–95% than Linux virtual memory, and the number of flash writes can be reduced by 38–93%.

1.2.2 FlashTier: Fast and Consistent Storage Cache

There are four contributions of FlashTier: (1) improved cache performance, (2) enhanced flash lifetime for write-intensive workloads, (3) reduced memory consumption for address translation across host and solid-state cache device, and (4) less runtime overhead for crash consistency and shorter recovery time for cached data.

FlashTier's performance benefits are derived from techniques designed for a solid-state cache (SSC) device custom built for caching: faster free space management for flash caching, and reduced crash-consistency overhead for address translation. FlashTier has a new free space management technique called silent eviction, which makes use of the state of a cached data block to evict it from SSC, and reduces the overhead for copying it. We also unify the address space by storing a single mapping in the SSC memory, which saves host memory and reduces cost of keeping two mapping consistent across power failures or crashes.

For write-intensive workloads, FlashTier achieves up to 168% performance improvement over consumer-grade SSDs using coarse hybrid address translation, and up to 52% performance improvement over high-end SSDs using fine-grained page-map address translation. We validate the performance improvement both in simulation and on a real hardware prototype. The SSC design also improves flash lifetime for caching write-intensive workloads by 40–60% compared to SSD caches.

FlashTier reduces the host memory consumption by avoiding double translation from disk to SSD logical block addresses to physical flash addresses. FlashTier instead directly translates disk logical block addresses to physical flash addresses. As a result of the unified address space, FlashTier significantly saves the host memory by up to 89% for address translation by not having mappings for SSD logical block addresses. It also does not require any host translation tables to reconstruct after a power failure or reboot and can recover a 100 GB

cache in less than 2.5 seconds.

1.3 Dissertation Organization

This dissertation describes the architecture and implementation of FlashVM and FlashTier. Chapter 2 provides background material on different memory technologies, block management and interface to SSDs, and application environments to use SSDs.

Chapter 3 describes the design, implementation and evaluation of FlashVM to extend memory to cheaper flash. We first present the design overview and then the implementation and evaluation of each of FlashVM's design techniques in detail. This chapter revises previous publications [97, 98].

Chapter 4 presents FlashTier design and implementation to improve the performance and lifetime of flash caching. We first describe FlashTier implementation in simulation and then on a real hardware platform. This chapter revises a previous publication [99].

Chapter 5 presents the lessons learned from transitioning the design of solid-state cache (SSC) device in FlashTier, evaluated earlier in simulation, to a hardware prototype on the OpenSSD Jasmine board [107]. This chapter documents our experiences and lessons from a previous publication [100].

Finally, Chapter 6 summarizes, and offers conclusions and future work directions.

2 BACKGROUND

In this chapter, we provide background on various technologies integral to this dissertation. First, we present a primer on flash memory technology, and discuss how it differs from DRAM, disk, and other emerging non-volatile memory technologies. Next, we review block management and standard interfaces to flash SSDs. Finally, we finish with an overview of application environments relevant to flash-backed virtual memory and caching.

2.1 Memory Technologies

Over the last decade, several non-volatile memory technologies have emerged to bridge the gap between DRAM and disk. Out of these, NAND flash has been the most notable technology ubiquitously available today. More than 21 *exabytes* of flash memory were manufactured worldwide in 2011, and is expected to further grow to 36 *exabytes* in 2012 [101]. Table 2.1 compares the price and performance characteristics of NAND flash memory with DRAM and disk. Flash price and performance are between DRAM and disk, and about five times cheaper than DRAM [26] and an order of magnitude faster than disk. Furthermore, flash power consumption (0.06 W when idle and 0.15–2 W when active) [45, 47] is significantly lower than both DRAM (4–5 W/DIMM) and disk (10–15 W). In addition, its persistence enables memory contents to survive crashes or power failures, and hence can improve cold-start performance. As a result, SSD-backed memory abstractions are getting popular in many environments

Device	**A**ccess Latency		**C**apacity	**P**rice	**E**ndurance	**P**ower
	Read	**W**rite	**B**ytes	**$**/GB	**E**rases	**W**atts
DRAM	50 ns	50 ns	8 GB	$5	∞	4-5 W/DIMM
SCM	100-200 ns	150-1000 ns	16-32 MB	NA	10^6	NA: 19.73 pJ/bit
Flash SSD	40-100 µs	60-200 µs	TB	$1	10^4	0.06-2 W
Disk	500-5000 µs	500-5000 µs	TB	$0.1	∞	10-15 W

Table 2.1: **M**emory Technologies: Price, performance, endurance, and power of DRAM, Storage Class Memory, NAND Flash SSDs and Disk. (MB: megabyte, GB: gigabyte, TB: terabyte, as of April 2013 [26]).

including workstations, enterprise, and network disk storage [98, 51, 28, 27, 80].

Internally, SSDs are comprised of multiple flash chips accessed in parallel. As a result, they provide scalable bandwidths limited mostly by the interface between the drive and the host machine (generally USB, SATA or PCIe) [5, 16]. In addition, SSDs provide access latencies orders of magnitude faster than traditional disks. For example, the FusionIO enterprise ioDrives provide 25 µs read latencies and up to 600 MB/sec throughput [31]. Consumer grade SSDs provide latencies down to 50-85 µs and bandwidths up to 250 MB/sec [45]. Systems must be designed to fully saturate the internal bandwidth of the devices, especially enterprise SSDs that support longer queues of I/O requests.

Another set of competing non-volatile memory technologies, commonly referred to as storage-class memory (SCM), provide persistence of flash and byte-addressability of DRAM. Phase-change memory (PCM) [2, 55], memristors [106] and STT-MRAM [88, 41] are promising implementations of SCM. These technologies are projected to achieve higher storage density than that of both DRAM and flash.

However, most of these technologies suffer from three major problems, which limit their ability to be commercially used over the next few years (see Table 2.1). First, their production has not scaled beyond single chips of a few hundreds of MBs capacity. The maximum size of PCM chips available today is of the order of 64 MB [2]. Second, currently available PCM is approximately twice slower than DRAM for reads, and about 20x slower for writes [2, 55]. Finally, most SCM technologies can not sustain more than 10^6 writes per cell. If it follows the same fate as that of flash, the endurance would degrade further with increased density and multi-cell encoding. As a result, most SCM technologies still do not contend with DRAM as memory replacement on the memory bus.

2.2 Taming Flash

Flash is a complex technology. There are two key properties of the flash medium that require special block management within flash SSDs to achieve high performance and reliability.

SSD Block Management. First, flash does not support *in-place writes*. Flash pages can

be read or written at a smaller granularity of less than 4 KB pages (takes up to 40-200 us). However, a full flash erase block composed of at least 64 4 KB flash pages must be *erased* (a lengthy operation, which can take up to 1 ms) before it can be written. This is because bits in a flash cell on a write operation can only be reset from 1s to 0s, and an erase operation is required to set them back to 1.

Second, each flash erase block can only be erased a finite number of times. For most consumer-grade multi-level cell (MLC) devices, the limit is 10,000 erase cycles per block, and this is further dropping with increases in flash density.

To support writing a block multiple times similar to a disk, SSD vendors implement within firmware a flash translation layer that uses *address mapping* to translate block addresses received from a host into physical locations in flash. This mapping allows a block to be written out-of-place to a pre-erased block rather than erasing and rewriting in-place. As a result, SSDs employ *garbage collection* to compact data and provide free, erased blocks for upcoming writes. The translation from this layer further raises three problems not present with disks: write amplification, low reliability, and aging.

Write amplification occurs when writing a single block causes the SSD to re-write multiple blocks, and leads to expensive read-modify-erase-write cycles (erase latency for a typical 256 KB flash block is as high as 1 millisecond) [5, 92]. Garbage collection is often a contributor to wear, as live data must be copied to make free blocks available. A recent study showed that more than 70% of the erasures on a full SSD were due to garbage collection [24]. Furthermore, SSDs exhibit aging after extensive use or near capacity usage, when there are fewer clean blocks available for writing [89, 92]. This can lead to performance degradation, as the device continuously copies data to clean pages.

SSD Block Interface. In addition to the traditional disk read/write interface, modern SSDs provide a *trim* command [102] for the OS to notify the device when blocks no longer contain valid data. This helps the SSD to garbage collect free blocks for improved performance and wear leveling. In its current form, trim acts like a hint that is only a performance optimization, not a contract for removing data from the device. Similar to disks, SSDs also expose a *flush*

command, which acts like a barrier: it guarantees that all enqueued commands are serviced and all blocks cached in device memory are durable.

This read/write/trim/flush block interface of SSDs is still very narrow and provides minimal coordination between block management within the OS/application and the SSD. In this thesis, we investigate more coordination through *new interfaces*, which result in better performance for free space management, well-defined consistency and durability semantics for writing and removing data within the SSD, and longer flash lifetime.

2.3 Emerging Applications for Virtual Memory and Caching

The emergence of SSDs enables new applications in various computing environments including mobile systems, laptops, desktops, servers and distributed clusters.

Mobile systems, laptops and desktops are usually constrained by cost, the number of DIMM slots, and DRAM power consumption. Memory-intensive workloads, such as image manipulation, video encoding, or even opening multiple tabs in a single web browser instance can consume hundreds of megabytes or gigabytes of memory in a few minutes of usage [23], thereby causing such systems to page. Furthermore, end users often run multiple programs, leading to competition for memory. Gigabytes of virtual memory backed with SSDs enables faster performance for memory intensive applications that scales with multiprogramming.

Enterprise servers and distributed systems such as key-value stores, web object caches, e-commerce platforms and picture stores, managing petabytes of on-disk data require both low latency reads and durable writes [28, 61, 9, 22, 33, 109]. Similarly, other infrastructures such as Amazon EC2 cloud compute servers accessing remote Amazon DynamoDB or S3 high-io storage instances [7], Twitter's compute servers using big data SSD-backed memory caches [109], or NFS servers accessing remote SAN storage require fast and reliable caching for large datasets. Fast and durable client-based caching is beneficial for virtual machine deployments, which are often constrained by the main memory capacities available on commonly deployed cheap servers [72, 32]. SSDs enable fast and persistent storage caching to provide the performance of flash with the cost of disks for such large datasets.

3 FLASHVM: LARGE AND CHEAP EXTENDED MEMORY

Flash memory is one of the largest changes to storage in recent history. Solid-state disks (SSDs), composed of multiple flash chips, provide the abstraction of a block device to the operating system similar to magnetic disks. This abstraction favors the use of flash as a replacement for disk storage due to its faster access speeds and lower energy consumption [5, 91, 114].

This chapter presents FlashVM, a system architecture and a core virtual memory subsystem built in the Linux kernel for managing flash-backed virtual memory. FlashVM extends a traditional system organization with dedicated flash for swapping virtual memory pages. Dedicated flash allows FlashVM software to use semantic information, such as the knowledge about free blocks, that is not available within an SSD. Furthermore, dedicating flash to virtual memory is economically attractive, because small quantities can be purchased for a few dollars. In contrast, disks ship only in large sizes at higher initial costs.

The design of FlashVM focuses on three aspects of flash: performance to embrace its unique characteristics, reliability for flash wear-out, and garbage collection of free VM pages. The remainder of the chapter is structured as follows. Section 3.1 describes the target environments and makes a case for FlashVM. Section 3.2 presents FlashVM design overview and challenges. We describe the design in Sections 3.3.1, covering performance; 3.3.2 covering reliability; and 3.3.3 on efficient garbage collection using the discard command. We evaluate the FlashVM design techniques in Section 3.4.

3.1 Motivation

Application working-set sizes have grown many-fold in the last decade, driving the demand for cost-effective mechanisms to improve memory performance. In this section, we motivate the use of flash-backed virtual memory by comparing it to DRAM and disk, and noting the workload environments that benefit the most.

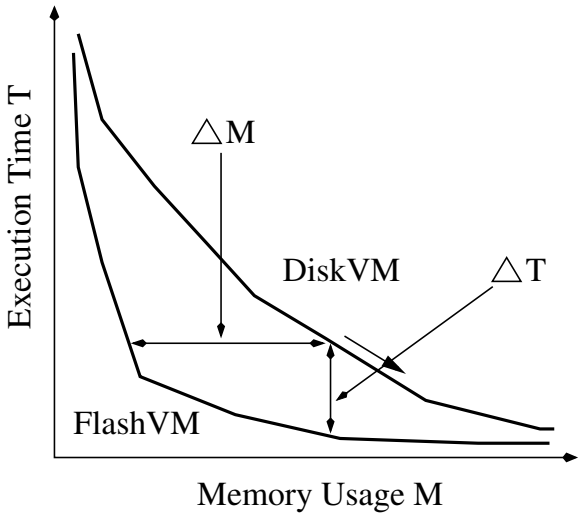


Figure 3.1: **C**ost/Benefit Analysis: Application execution time plot comparing the performance of disk and flash backed virtual memory with variable main memory sizes. ΔM is the memory savings to achieve the same performance as disk and ΔT is the performance improvement with FlashVM for the same memory size.

3.1.1 Why FlashVM?

Fast and cheap flash memory has become ubiquitous. More than 36 *exabytes* of flash memory were manufactured worldwide in 2012. Table 2.1 compares the price and performance characteristics of NAND flash memory with DRAM and disk. Flash price and performance are between DRAM and disk, and about five times cheaper than DRAM and an order of magnitude faster than disk. Furthermore, flash power consumption (0.06 W when idle and 0.15–2 W when active) is significantly lower than both DRAM (4–5 W/DIMM) and disk (13–18 W). These features of flash motivate its adoption as second-level memory between DRAM and disk.

Figure 3.1 illustrates a cost/benefit analysis in the form of two simplified curves (not to scale) showing the execution times for an application in two different systems configured with variable memory sizes, and either disk or flash for swapping. This figure shows two benefits to applications when they must page. First, FlashVM results in faster execution for approximately the same system price without provisioning additional DRAM, as flash is five times cheaper than DRAM (see Table 2.1). This performance gain is shown in Figure 3.1 as

ΔT along the vertical axis. Alternatively, we can also improve performance by adding more memory in the system, which is shown as improvement along the DiskVM curve. However, such addition of memory is not always feasible as systems are usually constrained by the number of DIMM slots, density and power requirements of configured DRAM. Second, a FlashVM system can achieve performance similar to swapping to disk with lower main memory requirements. This occurs because page faults are an order of magnitude faster with flash than disk: a program can achieve the same performance with less memory by faulting more frequently to FlashVM. This reduction in memory-resident working set is shown as ΔM along the horizontal axis.

However, the adoption of flash is fundamentally an economic decision, as performance can also be improved by purchasing additional DRAM or a faster disk. Thus, careful analysis is required to configure the balance of DRAM and flash memory capacities that is optimal for the target environment in terms of both price and performance.

3.1.2 Where FlashVM?

Both the price/performance gains for FlashVM are strongly dependent on the workload characteristics and the target environment. In this paper, we target FlashVM against the following workloads and environments:

Mobile Systems, Laptop and Desktops. Laptops and desktops are usually constrained with cost, the number of DIMM slots for DRAM modules and DRAM power-consumption. In these environments, the large capacity of disks is still desirable. Memory-intensive workloads, such as image manipulation, video encoding, or even opening multiple tabs in a single web browser instance can consume hundreds of megabytes or gigabytes of memory in a few minutes of usage [23], thereby causing the system to page. Furthermore, end users often run multiple programs, leading to competition for memory. FlashVM meets the requirements of such workloads and environments with faster performance that scales with increased multiprogramming.

Distributed Clusters. Data-intensive workloads such as virtualized services, key-value

stores and web caches have often resorted to virtual or distributed memory solutions. For example, the popular *memcached* [68] is used to increase the aggregate memory bandwidth. While disk access is too slow to support page faults during request processing, flash access times allow a moderate number of accesses. Flash SSDs provide acccess latencies of less than 200 µs and up to 100,000 operations per second, much faster than most mainline networks. Fast swapping can also benefit virtual machine deployments, which are often constrained by the main memory capacities available on commonly deployed cheap servers [72]. Virtual machine monitors can host more virtual machines with support for swapping out nearly the entire guest physical memory [32]. In such cluster scenarios, hybrid alternatives similar to FlashVM that incorporate DRAM and large amounts of flash are an attractive means to provide large memory capacities cheaply [33].

3.2 Design Overview

The FlashVM design, shown in Figure 3.2, consists of dedicated flash for swapping out virtual memory pages and changes to the Linux virtual memory hierarchy that optimize for the characteristics of flash. We target FlashVM against NAND flash, which has lower prices and better write performance than the alternative, NOR flash. We propose that future systems be built with a small multiple of DRAM size as flash that is attached to the motherboard for the express purpose of supporting virtual memory.

The FlashVM design leverages semantic information only available within the operating system, such as locality of memory references, page similarity and knowledge about deleted blocks, to provide high performance and reliability for flash management.

FlashVM Architecture. The FlashVM architecture targets dedicated flash for virtual memory paging. Dedicating flash for virtual memory has two distinct advantages over traditional disk-based swapping. First, dedicated flash is cheaper than traditional disk-based swap devices in price per byte only for small capacities required for virtual memory. A 4 GB MLC NAND flash chip costs less than $6, while the cheapest IDE/SCSI disk of similar size costs no less than $24 [25, 105]. Similarly, the more common SATA/SAS disks do not scale

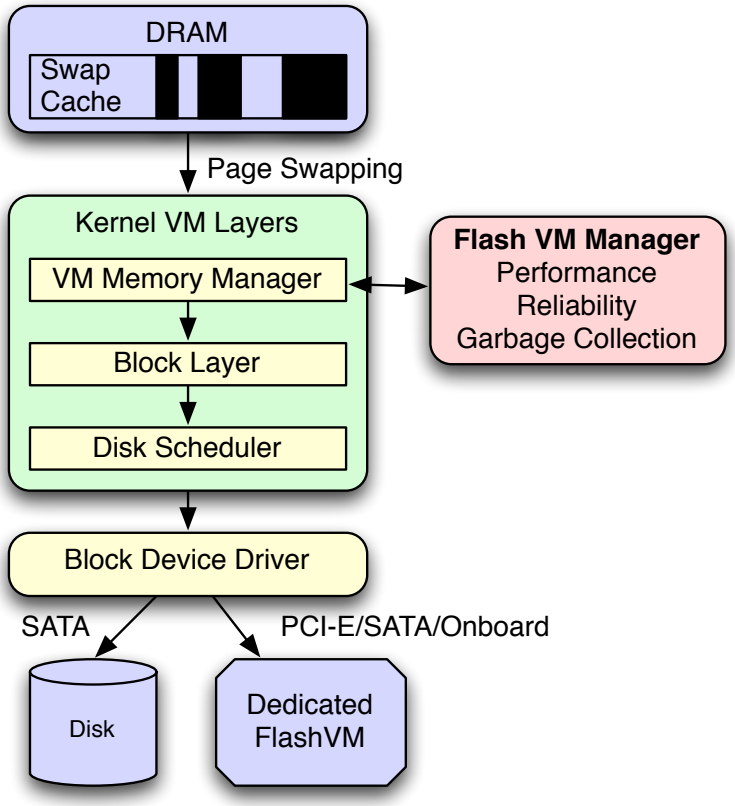


Figure 3.2: **F**lashVM Memory Hierarchy: FlashVM manager controls the allocation, read and write-back of pages swapped out from main memory. It hands the pages to the block layer for conversion into block I/O requests, which are submitted to the dedicated flash device.

down to capacities smaller than 36 GB, and even then are far more expensive than flash. Furthermore, the premium for managed flash, which includes a translation layer, as compared to raw flash chips is dropping rapidly as SSDs mature. Second, dedicating flash for virtual memory minimizes the interference between the file system I/O and virtual-memory paging traffic. We prototype FlashVM using MLC NAND flash-based solid-state disks connected over a SATA interface.

FlashVM Software. The FlashVM memory manager, shown in Figure 3.2, is an enhanced version of the memory management subsystem in the Linux 2.6.28 kernel. Since NAND flash is internally organized as a block device, the FlashVM manager enqueues the evicted pages at the block layer for scheduling. The block layer is responsible for the conversion of pages into block I/O requests submitted to the device driver. At a high-level, FlashVM manages the

non-ideal characteristics of flash and exploits its useful attributes. In particular, our design goals are:

- *High performance* by leveraging the unique performance characteristics of flash, such as fast random reads (discussed in Section 3.3.1).
- *Improved reliability* by reducing the number of page writes to the swap device (discussed in Section 3.3.2).
- *Efficient garbage collection* of free VM pages by delaying, merging, and virtualizing discard operations (discussed in Section 3.3.3).

The FlashVM implementation is not a singular addition to the Linux VM. As flash touches on many aspects of performance, FlashVM modifies most components of the Linux virtual memory hierarchy, including the swap-management subsystem, the memory allocator, the page scanner, the page-replacement and prefetching algorithms, the block layer and the SCSI subsystem. In the next section, we identify and describe our changes to each of these subsystems for achieving the different FlashVM design goals.

3.3 Design and Implementation

This section discusses the FlashVM implementation to improve performance, reliability, and to provide efficient garbage collection.

3.3.1 FlashVM Performance

Challenges. The virtual-memory systems of most operating systems were developed with the assumption that disks are the only swap device. While disks exhibit a range of performance, their fundamental characteristic is the speed difference between random and sequential access. In contrast, flash devices have a different set of performance characteristics, such as fast random reads, high sequential bandwidths, low access and seek costs, and slower writes than reads. For each code path in the VM hierarchy affected by these differences between flash and

disk, we describe our analysis for tuning parameters and our implementation for optimizing performance with flash. We analyze three VM mechanisms: *page pre-cleaning*, *page clustering* and *disk scheduling*, and re-implement *page scanning* and *prefetching* algorithms.

Page Write-Back

Swapping to flash changes the performance of writing back dirty pages. Similar to disk, random writes to flash are costlier than sequential writes. However, random reads are inexpensive, so write-back should optimize for write locality rather than read locality. FlashVM accomplishes this by leveraging the *page pre-cleaning* and *clustering* mechanisms in Linux to reduce page write overheads.

Pre-cleaning. Page pre-cleaning is the act of eagerly swapping out dirty pages before new pages are needed. The Linux page-out daemon *kswapd* runs periodically to write out 32 pages from the list of inactive pages. The higher write bandwidth of flash allows FlashVM write pages more aggressively, and without competing file system traffic, and use more I/O bandwidth.

Thus, we investigate writing more pages at a time to achieve sequential write performance on flash. For disks, pre-cleaning more pages interferes with high-priority reads to service a fault. However, with flash, the lower access latency and higher bandwidths enable more aggressive pre-cleaning without affecting the latency for handling a page-fault.

Clustering. The Linux clustering mechanism assigns locations in the swap device to pages as they are written out. To avoid random writes, Linux allocates *clusters*, which are contiguous ranges of *page slots*. When a cluster has been filled, Linux scans for a free cluster from the start of the swap space. This reduces seek overheads on disk by consolidating paging traffic near the beginning of the swap space.

We analyze FlashVM performance for a variety of cluster sizes from 8 KB to 4096 KB. In addition, for clusters at least the size of an erase block, we align clusters with erase-block boundaries to ensure minimum amount of data must be erased.

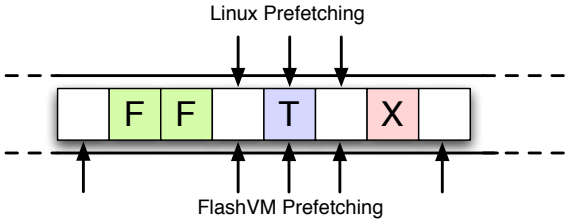


Figure 3.3: **V**irtual Memory Prefetching: Linux reads-around an aligned block of pages consisting of the target page and delimited by free or bad blocks to minimize disk seeks. FlashVM skips the free and bad blocks by seeking to the next allocated page and reads all valid pages. (T, F and X represent target, free and bad blocks on disk respectively; unfilled boxes represent the allocated pages).

Page Scanning

A virtual memory system must ensure that the rate at which it selects pages for eviction matches the write bandwidth of the swap device. Pages are selected by two code paths: memory reclaim during page allocation; and the page-out daemon that scan the inactive page list for victim pages. The Linux VM subsystem balances the rate of scanning with the rate of write-back to match the bandwidth of the swap device. If the scanning rate is too high, Linux throttles page write-backs by waiting for up to 20–100 milliseconds or until a write completes. This timeout, appropriate for disk, is more than two orders of magnitude greater than flash access latencies.

FlashVM controls write throttling at a much finer granularity of a system *jiffy* (one clock tick). Since multiple page writes in a full erase block on flash take up to two milliseconds, FlashVM times-out for about one millisecond on our system. These timeouts do not execute frequently, but have a large impact on the average page fault latency [97]. This enables FlashVM to maintain higher utilization of paging bandwidth and speeds up the code path for write-back when reclaiming memory.

Prefetching on Page Fault

Operating systems prefetch pages after a page fault to benefit from the sequential read bandwidth of the device [59]. The existing Linux prefetch mechanism reads in up to 8 pages contiguous on disk around the target page. Prefetching is limited by the presence of free or

bad page slots that represent bad blocks on disk. As shown in Figure 3.3, these page slots delimit the start or the end of the prefetched pages. On disk, this approach avoids the extra cost of seeking around free and bad pages, but often leads to fetching fewer than 8 pages.

FlashVM leverages fast random reads on flash with two different prefetching mechanisms. First, FlashVM seeks over the free/bad pages when prefetching to retrieve a full set of valid pages. Thus, the fast random access of flash medium enables FlashVM to bring in more pages with spatial locality than native Linux.

Fast random access on flash also allows prefetching of more distant pages with temporal locality, such as stride prefetching. FlashVM records the offsets between the current target page address and the last two faulting addresses. Using these two offsets, FlashVM computes the strides for the next two pages expected to be referenced in the future. Compared to prefetching adjacent pages, stride prefetching reduces memory pollution by reading the pages that are more likely to be referenced soon.

We implement stride prefetching to work in conjunction with contiguous prefetching: FlashVM first reads pages contiguous to the target page and then prefetches stride pages. We find that fetching too many stride pages increases the average page fault latency, so we limit the stride to two pages. These two prefetching schemes result in a reduction in the number of page faults and improve the total execution time for paging.

Disk Scheduling

The Linux VM subsystem submits page read and write requests to the block-layer I/O scheduler. The choice of the I/O scheduler affects scalability with multiprogrammed workloads, as the scheduler selects the order in which requests from different processes are sent to the swap device.

Existing Linux I/O schedulers optimize performance by (i) merging adjacent requests, (ii) reordering requests to minimize seeks and to prioritize requests, and (iii) delaying requests to allow a process to submit new requests. Work-conserving schedulers, such as the NOOP and deadline schedulers in Linux, submit pending requests to the device as soon as the prior

request completes. In contrast, non-work-conserving schedulers may delay requests for up to 2–3 ms to wait for new requests with better locality or to distribute I/O bandwidth fairly between processes [49]. However, these schedulers optimize for the performance characteristics of disks, where seek is the dominant cost of I/O. We therefore analyze the impact of different I/O schedulers on FlashVM.

The Linux VM system tends to batch multiple read requests on a page fault for prefetching, and multiple write requests for clustering evicted pages. Thus, paging traffic is more regular than file system workloads in general. Further, delaying requests for locality can lead to lower device utilization on flash, where random access is only a small component of the page transfer cost. Thus, we analyze the performance impact of work conservation when scheduling paging traffic for FlashVM.

3.3.2 FlashVM Reliability

Challenges. As flash geometry shrinks and multi-level cell (MLC) flash technology packs more bits into each memory cell, the prices of flash devices have dropped significantly. Unfortunately, so has the *erasure limit* per flash block. A single flash block can typically undergo between 10,000 and 100,000 erase cycles, before it can no longer reliably store data. Modern SSDs and flash devices use internal wear-leveling to spread writes across all flash blocks. However, the bit error rates of these devices can still become unacceptably high once the erasure limit is reached [71]. As the virtual memory paging traffic may stress flash storage, FlashVM specially manages page writes to improve device reliability. It exploits the information available in the OS about the state and content of a page by employing *page sampling* and *page sharing* respectively. FlashVM aims to reduce the number of page writes and prolong the lifetime of the flash device dedicated for swapping.

Page Sampling

Linux reclaims free memory by evicting inactive pages in a least-recently-used order. Clean pages are simply added to the free list, while reclaiming dirty pages requires writing them

back to the swap device.

FlashVM modifies the Linux page replacement algorithm by prioritizing the reclaim of younger *clean* pages over older *dirty* pages. While scanning for pages to reclaim, FlashVM skips dirty pages with a probability dependent on the rate of pre-cleaning. This policy increases the number of clean pages that are reclaimed during each scan, and thus reduces the overall number of writes to the flash device.

The optimal rate for sampling dirty pages is strongly related to the memory reference pattern of the application. For applications with read-mostly page references, FlashVM can find more clean pages to reclaim. However, for applications that frequently modify many pages, skipping dirty pages for write-back leads to more frequent page faults, because younger clean pages must be evicted.

FlashVM addresses workload variations with adaptive page sampling: the probability of skipping a dirty page also depends on the write rate of the application. FlashVM predicts the average write rate by maintaining a moving average of the time interval $\mathtt{t_n}$ for writing n dirty pages. When the application writes to few pages and $\mathtt{t_n}$ is large, FlashVM more aggressively skips dirty pages. For applications that frequently modify many pages, FlashVM reduces the page sampling probability unless n pages have been swapped out. The balance between the rate of page sampling and page writes is adapted to provide a smooth tradeoff between device lifetime and application performance.

Page Sharing

The Linux VM system writes back pages evicted from the LRU inactive list without any knowledge of page content. This may result in writing many pages to the flash device that share the same content. Detecting identical or similar pages may require heavyweight techniques like explicitly tracking changes to each and every page by using transparent page sharing [12] or content-based page sharing by maintaining hash signatures for all pages [39]. These techniques reduce the memory-resident footprint and are orthogonal to the problem of reducing the number of page write-backs.

We implement a limited form of content-based sharing in FlashVM by detecting the swap-out of *zero pages* (pages that contain only zero bytes). Zero pages form a significant fraction of the memory-footprint of some application workloads [39]. FlashVM intercepts paging requests for all zero pages. A swap-out request sets a zero flag in the corresponding page slot in the swap map, and skips submitting a block I/O request. Similarly, a swap-in request verifies the zero flag, which if found set, allocates a zero page in the address space of the application. This extremely lightweight page sharing mechanism saves both the memory allocated for zero pages in the main-memory swap cache and the number of page write-backs to the flash device.

3.3.3 FlashVM Garbage Collection

Challenges. Flash devices cannot overwrite data in place. Instead, they must first erase a large flash block (128–512 KB), a slow operation, and then write to pages within the erased block. Lack of sufficient pre-erased blocks may result in copying multiple flash blocks for a single page write to: (i) replenish the pool of clean blocks, and (ii) ensure uniform wear across all blocks. Therefore, flash performance and overhead of wear management are strongly dependent on the number of clean blocks available within the flash device. For example, high-end enterprise SSDs can suffer up to 85% drop in write performance after extensive use [89, 92]. As a result, efficient garbage collection of clean blocks is necessary for flash devices, analogous to the problem of segment cleaning for log-structured file systems [95].

For virtual memory, sustained paging can quickly *age* the dedicated flash device by filling up all free blocks. When FlashVM overwrites a block, the device can reclaim the storage previously occupied by that block. However, only the VM system has knowledge about empty (free) page clusters. These clusters consist of page slots in the swap map belonging to terminated processes, dirty pages that have been read into the memory and all blocks on the swap device after a reboot. Thus, a flash device that implements internal garbage collection or wear-leveling may unnecessarily copy stale data, reducing performance and reliability when not informed about invalid pages.

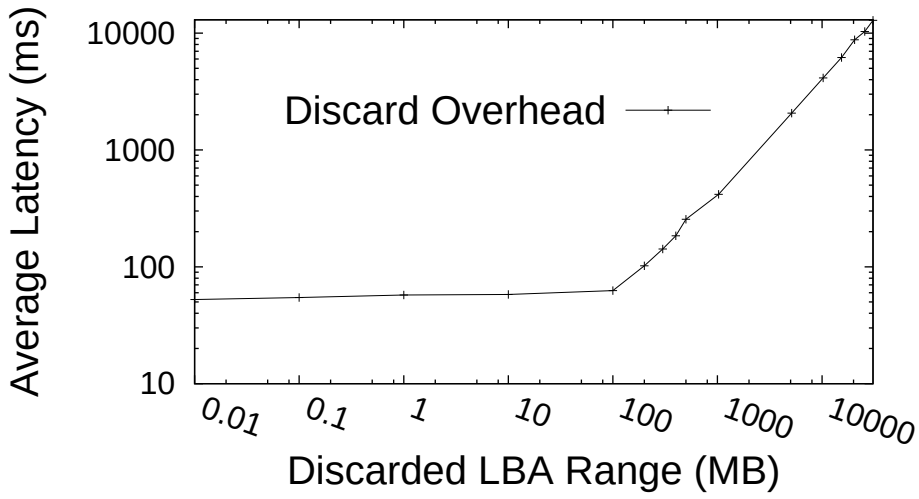


Figure 3.4: **D**iscard Overhead: Impact of the number of blocks discarded on the average latency of a single discard command. Both x-axis and y-axis are log-scale.

FlashVM addresses this problem by explicitly notifying the flash device of such pages by using the *discard* command (also called *trim* introduced in the recent SATA SSDs [102]). The discard command has the following semantics:

discard(*dev*, *rangelist*[(sector, nsectors), ...])

where *rangelist* is the list of logical block address ranges to be discarded on the flash device *dev*. Each block range is represented as a pair of the starting sector address and the number of following sectors.

Free blocks can be discarded offline by first flushing all in-flight read/write requests to the flash device, and then wiping the requested logical address space. However, offline discard typically offers very coarse-grain functionality, for example in the form of periodic disk scrubbing or disk format operations [73]. Therefore, FlashVM employs online cleaning that discards a smaller range of free flash page clusters at runtime.

Linux implements rudimentary support for online cleaning in recent kernel versions starting

in 2.6.28. When it finds 1 MB of free contiguous pages, it submits a single discard request for the corresponding page cluster. However, Linux does not fully support discard yet: the block layer breaks discard requests into smaller requests of no more than 256 sectors, while the ATA disk driver ignores them. Thus, the existing Linux VM is not able to actually discard the free page clusters. FlashVM instead bypasses the block and ATA driver layers and sends discard commands directly to the flash device through the SCSI layer [62, 64]. Thus, FlashVM has the ability to discard any number of sectors in a single command.

We next present an experimental analysis of the cost of discard on current flash devices. Based on these results, which show that discard is expensive, we describe two different techniques that FlashVM uses to improve the performance of garbage collection: *merged discard* and *dummy discard*.

Discard Cost

We measure the latency of discard operations on the OCZ-Vertex SSD, which uses the Indilinx flash controller used by many SSD manufacturers. Figure 3.4 shows the overheads for discard commands issued over block address ranges of different sizes. Based on Figure 3.4, we infer the cost of a single discard command for cleaning B flash blocks in one or more address ranges, each having an average utilization u of valid (not previously cleaned) pages:

$$\mathtt{cost}_{\mathtt{M}} = \begin{cases} \mathtt{c_o} & \text{if } \mathtt{B} \leqslant \mathtt{B_o} \\ \mathtt{c_o} + \mathtt{m} \cdot \mathtt{u} \cdot (\mathtt{B} - \mathtt{B_o}) & \text{otherwise} \end{cases}$$

In this equation, $\mathtt{c_o}$ is the fixed cost of discarding up to $\mathtt{B_o}$ blocks and m is the marginal cost of discarding each additional block. *Interestingly, the fixed cost of a single discard command is 55 milliseconds!* We speculate that this overhead occurs because the SSD controller performs multiple block erase operations on different flash channels when actually servicing a discard command [63]. The use of an on-board RAM buffer conceals the linear increase only up to a range of $\mathtt{B_o}$ blocks lying between 10–100 megabytes.

The cost of discard is exacerbated by the effect of command queuing: the ATA specification

defines the discard commands as *untagged*, requiring that every discard be followed by an I/O barrier that stalls the request queue while it is being serviced. Thus, the long latency of discards requires that FlashVM optimize the use of the command, as the overhead incurred may outweigh the performance and reliability benefits of discarding free blocks.

Merged Discard

The first optimization technique that FlashVM uses is *merged discard*. Linux limits the size of each discard command sent to the device to 128 KB. However, as shown in Figure 3.4, discards get cheaper per byte as range sizes increase. Therefore, FlashVM opportunistically discards larger block address ranges. Rather than discard pages on every scan of the swap map, FlashVM defers the operation and batches discards from multiple scans. It discards the largest possible range list of free pages up to a size of 100 megabytes in a single command.

This approach has three major benefits. First, delaying discards reduces the overhead of scanning the swap map. Second, merging discard requests amortizes the fixed discard cost c_o over multiple block address ranges. Third, FlashVM merges requests for fragmented and non-contiguous block ranges. In contrast, the I/O scheduler only merges contiguous read/write requests.

Dummy Discard

Discard is only useful when it creates free blocks that can later be used for writes, similar to cleaning cold segments in log-structured file systems [95]. Overwriting a block *also* causes the SSD to discard the old block contents, but without paying the high fixed costs of the discard command. Furthermore, overwriting a free block removes some of the benefit of discarding to maintain a pool of empty blocks. Therefore, FlashVM implements *dummy discard* to avoid a discard operation when unnecessary.

Dummy discard elides a discard operation if the block is likely to be overwritten soon. This operation implicitly informs the device that the old block is no longer valid and can be asynchronously garbage collected without incurring the fixed cost of a discard command.

As FlashVM only writes back page clusters that are integral multiples of erase-block units, no data from partial blocks needs to be relocated. Thus, the cost for a dummy discard is effectively zero.

Unlike merged discards, dummy discards do not make new clean blocks available. Rather, they avoid an ineffective discard, and can therefore only replace a fraction of all discard operations. FlashVM must therefore decide when to use each of the two techniques. Ideally, the number of pages FlashVM discards using each operation depends on the available number of clean blocks, the ratio of their costs and the rate of allocating free page clusters. Upcoming and high-end enterprise SSDs expose the number of clean blocks available within the device [89]. In the absence of such functionality, FlashVM predicts the rate of allocation by estimating the expected time interval $\mathtt{t_s}$ between two successive scans for finding a free page cluster. When the system scans frequently, recently freed blocks are overwritten soon, so FlashVM avoids the extra cost of discarding the old contents. When scans occur rarely, discarded clusters remain free for an extended period and benefit garbage collection. Thus, when $\mathtt{t_s}$ is small, FlashVM uses dummy discards, and otherwise applies merged discards to a free page cluster in the swap map.

3.3.4 Summary

FlashVM architecture improves performance, reliability and garbage collection overheads for paging to dedicated flash. Some of the techniques incorporated in FlashVM, such as zero-page sharing, also benefit disk-backed virtual memory. However, the benefit of sharing is more prominent for flash, as it provides both improved performance and reliability.

While FlashVM is designed for managed flash, much of its design is applicable to unmanaged flash as well. In such a system, FlashVM would take more control over garbage collection. With information about the state of pages, it could more effectively clean free pages without an expensive discard operation. Finally, this design avoids the cost of storing persistent mappings of logical block addresses to physical flash locations, as virtual memory is inherently volatile.

3.4 Evaluation

The implementation of FlashVM entails two components: changes to the virtual memory implementation in Linux and dedicated flash for swapping. We implement FlashVM by modifying the memory management subsystem and the block layer in the x86-64 Linux 2.6.28 kernel. We focus our evaluation on three key questions surrounding these components:

- How much does the FlashVM architecture of *dedicated flash* for virtual memory improve performance compared to traditional disk-based swapping?
- Does FlashVM software design improve *performance*, *reliability* via write-endurance and *garbage collection* for virtual memory management on flash?
- Is FlashVM a *cost-effective* approach to improving system price/performance for different real-world application workloads?

We first describe our experimental setup and methodology and then present our evaluation to answer these three questions in Section 3.4.2, 3.4.3 and 3.4.4 respectively. We answer the first question by investigating the benefits of dedicating flash for paging in Section 3.4.2. In Section 3.4.3 and 3.4.4, we isolate the impact of FlashVM software design by comparing against the native Linux VM implementation.

3.4.1 Methodology

System and Devices. We run all tests on a 2.5 GHz Intel Core 2 Quad system configured with 4 GB DDR2 DRAM and 3 MB L2 cache per core, although we reduce the amount of memory available to the OS for our tests, as and when mentioned. We compare four storage devices: an IBM first generation SSD, a trim-capable OCZ-Vertex SSD, an Intel X-25M second generation SSD, and a Seagate Barracuda 7200 RPM disk, all using native command queuing. Device characteristics are shown in Table 3.1.

Application Workloads. We evaluate FlashVM performance with four memory-intensive application workloads with varying working set sizes:

Device	**S**equential (MB/s)		**R**andom 4K-IO/s		**L**atency
	Read	**W**rite	**R**ead	**W**rite	ms
Seagate Disk	80	68	120-300/s		4-5
IBM SSD	69	20	7K/s	66/s	0.2
OCZ SSD	230	80	5.5K/s	4.6K/s	0.2
Intel SSD	250	70	35K/s	3.3K/s	0.1

Table 3.1: **D**evice Characteristics: First-generation IBM SSD is comparable to disk in read bandwidth but excels for random reads. Second-generation OCZ-Vertex and Intel SSDs provide both faster read/write bandwidths and IOPS. Write performance asymmetry is more prominent in first-generation SSDs.

1. *ImageMagick* 6.3.7, resizing a large JPEG image by 500%,

2. *Spin* 5.2 [103], an LTL model checker for testing mutual exclusion and race conditions with a depth of 10 million states,

3. *pseudo-SpecJBB*, a modified SpecJBB 2005 benchmark to measure execution time for 16 concurrent data warehouses with 1 GB JVM heap size using Sun JDK 1.6.0,

4. *memcached* 1.4.1 [68], a high-performance object caching server bulk-storing or looking-up 1 million random 1 KB key-value pairs.

All workloads have a virtual memory footprint large enough to trigger paging and reach steady state for our analysis. For all our experiments, we report results averaged over five different runs. While we tested with all SSDs, we mostly present results for the second generation OCZ-Vertex and Intel SSDs for brevity.

3.4.2 Dedicated Flash

We evaluate the benefit of dedicating flash to virtual memory by: (i) measuring the costs of sharing storage with the file system, which arise from scheduling competing I/O traffic, and (ii) comparing the scalability of virtual memory with traditional disk-based swapping.

Read/Write Interference

With disk, the major cost of interference is the seeks between competing workloads. With an SSD, however, seek cost is low and the cost of interference arises from interleaving reads and writes from the file and VM systems. Although this cost occurs with disks as well, it is dominated by the overhead of seeking. We first evaluate the performance loss from interleaving, and then measure the actual amount of interleaving with FlashVM.

We use a synthetic benchmark that reads or writes a sequence of five contiguous blocks. Figure 3.5a shows I/O performance as we interleave reads and writes for disk, IBM SSD and Intel SSD. For disk, I/O performance drops from its sequential read bandwidth of 80 MB/s to 8 MB/s when the fraction of interleaved writes reaches 60% because the drive head has to be repositioned between read and write requests. On flash, I/O performance also degrades as the fraction of writes increase: IBM and Intel SSDs performance drops by 10x and 7x respectively when 60 percent of requests are writes. Thus, interleaving can severely reduce system performance.

These results demonstrate the potential improvement from dedicated flash, because, unlike the file system, the VM system avoids interleaved read and write requests. To measure this ability, we traced the block I/O requests enqueued at the block layer by the VM subsystem using Linux *blktrace*. Page read and write requests are governed by prefetching and page-out operations, which batch up multiple read/write requests together. On analyzing the average length of read request streams interleaved with write requests for ImageMagick and Spin, we found that FlashVM submits long strings of read and write requests. The average length of read streams ranges between 138–169 I/O requests, and write streams are between 170–230 requests. Thus, the FlashVM system architecture benefits from dedicating flash without interleaved reads and writes from the file system.

Scaling Virtual Memory

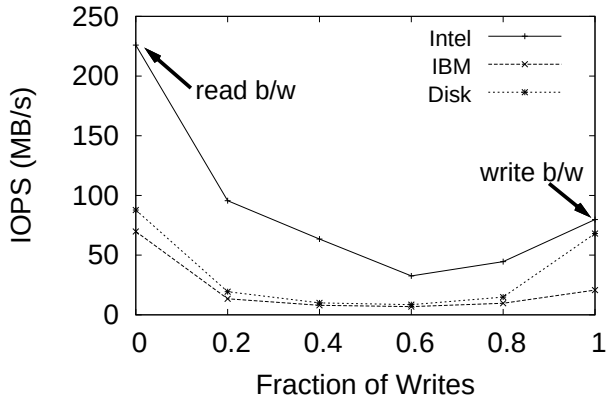
Unlike flash, dedicating a disk for swapping does not scale with multiple applications contending for memory. This scalability manifests in two scenarios: increased throughput as the number

of threads or programs increases, and decreased interference between programs competing for memory.

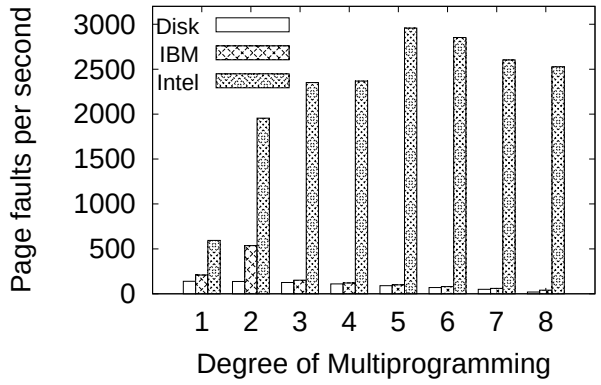
Multiprogramming. On a dedicated disk, competing programs degenerate into random page-fault I/O and high seek overheads. Figure 3.5b compares the paging throughput on different devices as we run multiple instances of ImageMagick. Performance, measured by the rate of page faults served per second, degrades for both disk and the IBM SSD with as few as 3 program instances, leading to a CPU utilization of 2–3%. For the IBM SSD, performance falls largely due to an increase in the random write traffic, which severely degrades its performance.

In contrast, we find improvement in the effective utilization of paging bandwidth on the Intel SSD with an increase in concurrency. At 5 instances, paging traffic almost saturates the device bandwidth: for each page fault FlashVM prefetches an additional 7 pages, so it reads 96 MB/s to service 3,000 page faults per second. In addition, it writes back a proportional but lower number of pages. Above 5 instances of ImageMagick, the page fault service rate drops because of increased congestion for paging traffic: CPU utilization falls from 54% with 5 concurrent programs to 44% for 8 programs, and write traffic nears the bandwidth of the device. Nevertheless, these results demonstrate that performance scales significantly as multiprogramming increases on flash when compared to disk. We find similar increase in paging throughput on dedicated flash for multithreaded applications like memcached. FlashVM performance and device utilization increase when more threads generate more simultaneous requests. This is much the same argument that exists for hardware multithreading to increase parallelism in the memory system.

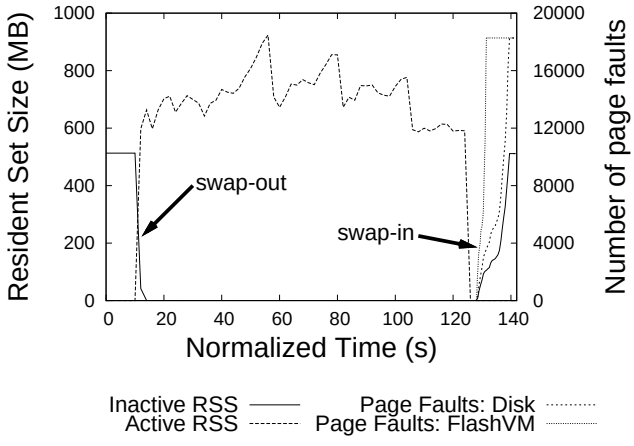
Response Time. The second scalability benefit of dedicated flash is faster response time for demand paging when multiple applications contend for memory. Figure 3.5c depicts the phenomena frequently observed on desktops when switching to inactive applications. We model this situation with two processes each having working-set sizes of 512 MB and 1.5 GB, that contend for memory on a system configured with 1 GB of DRAM. The curves show the resident set sizes (the amount of physical memory in use by each process) and the aggregate number of page faults in the system over a time interval of 140 seconds. The first process is



(a) Read/Write Interference



(b) Paging Throughput



(c) Response Time

Figure 3.5: **D**edicated Flash: Impact of dedicating flash for VM on performance for read/write interference with file system traffic, paging throughput for multiprogrammed workloads and response time for increased memory contention.

active for the first 10 seconds and is then swapped out by the Linux VM to accommodate the other process. When 120 seconds have elapsed, the second process terminates and the first process resumes activity.

Demand paging the first process back into memory incurs over 16,000 page faults. With disk, this takes 11.5 seconds and effectively prevents all other applications from accessing the disk. In contrast, resuming the first process takes only 3.5 seconds on flash because of substantially lower flash access latency. Thus, performance degrades much more acceptably with dedicated flash than traditional disk-based swapping, leading to better scalability as the number of processes increase.

3.4.3 FlashVM Software Evaluation

FlashVM enhances the native Linux virtual memory system for improved performance, reliability and garbage collection. We first analyze our optimizations to the existing VM mechanisms required for improving flash performance, followed by our enhancements for wear management and garbage collection of free blocks.

Performance Analysis

We analyze the performance of different codes paths that impact the paging performance of FlashVM.

Page pre-cleaning. Figure 3.6a shows the performance of FlashVM for ImageMagick, Spin and memcached as we vary the number of pages selected for write-back (pre-cleaned) on each page fault. Performance is poor when only two pages are written back because the VM system frequently scans the inactive list to reclaim pages. However, we find that performance does not improve when pre-cleaning more than 32 pages, because the overhead of scanning is effectively amortized at that point.

Page clustering. Figure 3.6b shows FlashVM performance as we vary the cluster size, the number of pages allocated contiguously on the swap device, while keeping pre-cleaning

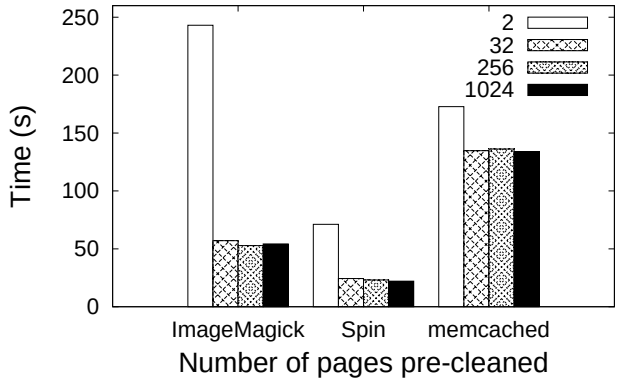
constant at 32 pages. When only two pages are allocated contiguously (cluster size is two), overhead increases because the VM system wastes time finding free space. Large cluster sizes lead to more sequential I/O, as pages are allocated sequentially within a cluster. However, our results show that above 32 page clusters, performance again stabilizes. This occurs because 32 pages, or 128 KB, is the size of a flash erase block and is enough to obtain the major benefits of sequential writes on flash. We tune FlashVM with these optimal values for pre-cleaning and cluster sizes for all our further experiments.

Congestion control. We evaluate the performance of FlashVM congestion control by comparing it against native Linux on single- and multi-threaded workloads. We separately measured the performance of changing the congestion timeout for the page allocator and for both the page allocator and the *kswapd* page-out daemon for ImageMagick. With the native Linux congestion control timeout tuned to disk access latencies, the system idles even when there is no congestion.

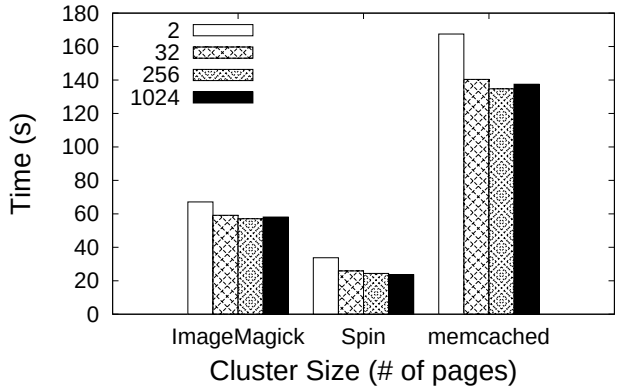
For single-threaded programs, reducing the timeout for the page allocator from 20ms to 1ms improved performance by 6%, and changing the timeout for *kswapd* in addition leads to a 17% performance improvement. For multithreaded workloads, performance improved 4% for page allocation and 6% for both page allocation and the *kswapd*. With multiple threads, the VM system is less likely to idle inappropriately, leading to lower benefits from a reduced congestion timeout. Nevertheless, FlashVM configures lower timeouts, which better match the latency for page access on flash.

Prefetching. Along the page-fault path, FlashVM prefetches more aggressively than Linux by reading more pages around the faulting address and fetching pages at a stride offset. Table 3.2 shows the benefit of these two optimizations for ImageMagick. The table lists the number of page faults and performance as we vary the number of pages read-ahead for FlashVM prefetching against native Linux prefetching, both on the Intel SSD.

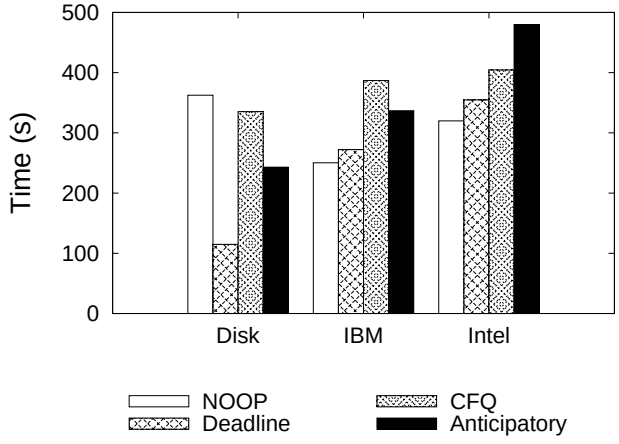
We find that FlashVM outperforms Linux for all values of read-ahead. The reduction in page faults improves from 15% for two pages to 35% for 16 pages, because of an increase in the difference between the number of pages read for native Linux and FlashVM. However,



(a) Page pre-cleaning



(b) Page clustering



(c) Disk Scheduling

Figure 3.6: **P**erformance Analysis: Impact of page pre-cleaning, page clustering and disk scheduling on FlashVM performance for different application workloads.

Read-Ahead (# of pages)	**N**ative		**S**tride	
	PF	**T**ime	**P**F	**T**ime
2	139K	103.2	118K / 15%	88.5 / 14%
4	84K	96.3	70K / 17%	85.7 / 11%
8	56K	91.5	44K / 21%	85.1 / 7%
16	43K	89.0	28K / 35%	83.5 / 6%

Table 3.2: **V**M Prefetching: Impact of native Linux and FlashVM prefetching on the number of page faults and application execution time for ImageMagick. (PF is number of hard page faults in thousands, Time is elapsed time in seconds, and percentage reduction and speedup are shown for the number of page faults and application execution time respectively.)

the speedup decreases because performance is lost to random access that results in increased latency per page fault. More sophisticated application-directed prefetching can provide additional benefits by exploiting a more accurate knowledge of the memory reference patterns and the low seek costs on flash.

Disk Scheduling. FlashVM depends on the block layer disk schedulers for merging or re-ordering I/O requests for efficient I/O to flash. Linux has four standard schedulers, which we compare in Figure 3.6c. For each scheduler, we execute 4 program instances concurrently and report the completion time of the last program. We scale the working set of the program instances to ensure relevant comparison on each individual device, so the results are not comparable across devices.

On disk, the NOOP scheduler, which only merges adjacent requests before submitting them to the block device driver in FIFO order, performs worst, because it results in long seeks between requests from different processes. The deadline scheduler, which prioritizes synchronous page faults over asynchronous writes, performs best. The other two schedulers, CFQ and anticipatory, insert delays to minimize seek overheads, and have intermediate performance.

In contrast, for both flash devices the NOOP scheduler outperforms all other schedulers, outperforming CFQ and anticipatory scheduling by as much as 35% and the deadline scheduler by 10%. This occurs because there is no benefit to localizing seeks on an SSD. We find

that average page access latency measured for disk increases linearly from 1 to 6 ms with increasing seek distance. In contrast, for both SSDs, seek time is constant and less than 0.2 ms even for seek distances up to several gigabytes. So, the best schedule for SSDs is to merge adjacent requests and queue up as many requests as possible to obtain the maximum bandwidth. We find that disabling delaying of requests in the anticipatory scheduler results in a 22% performance improvement, but it is still worse than NOOP. Thus, non work-conserving schedulers are not effective when swapping to flash, and scheduling as a whole is less necessary. For the remaining tests, we use the NOOP scheduler.

Wear Management

FlashVM reduces wear-out of flash blocks by write reduction using dirty page sampling and zero-page sharing.

Page Sampling. For ImageMagick, uniformly skipping 1 in 100 dirty pages for write back results in up to 12% reduction in writes but a 5% increase in page faults and a 7% increase in the execution time. In contrast, skipping dirty pages aggressively only when the program has a lower write rate better prioritizes the eviction of clean pages. For the same workload, adaptively skipping 1 in 20 dirty pages results in a 14% write reduction without any increase in application execution time. Thus, adaptive page sampling better reduces page writes with less affect on application performance.

Page Sharing. The number of zero pages swapped out from the inactive LRU list to the flash device is dependent on the memory-footprint of the whole system. Memcached clients bulk-store random keys, leading to few empty pages and only 1% savings in the number of page writes with zero-page sharing. In contrast, both ImageMagick and Spin result in substantial savings. ImageMagick shows up to 15% write reduction and Spin swaps up to 93% of zero pages. We find that Spin pre-allocates a large amount of memory and zeroes it down before the actual model verification phase begins. Zero-page sharing improves both the application execution time as well as prolongs the device lifetime by reducing the number of page writes.

Garbage Collection

FlashVM uses *merged* and *dummy* discards to optimize garbage collection of free VM pages on flash. We compare the performance of garbage collection for FlashVM against native Linux VM on an SSD. Because Linux cannot currently execute discards, we instead collect block-level I/O traces of paging traffic for different applications. The block layer breaks down the VM discard I/O requests into 128 KB discard commands, and we emulate FlashVM by merging multiple discard requests or replacing them with equivalent dummy discard operations as described in Section 3.3.3. Finally, we replay the processed traces on an aged trim-capable OCZ-Vertex SSD and record the total trace execution time.

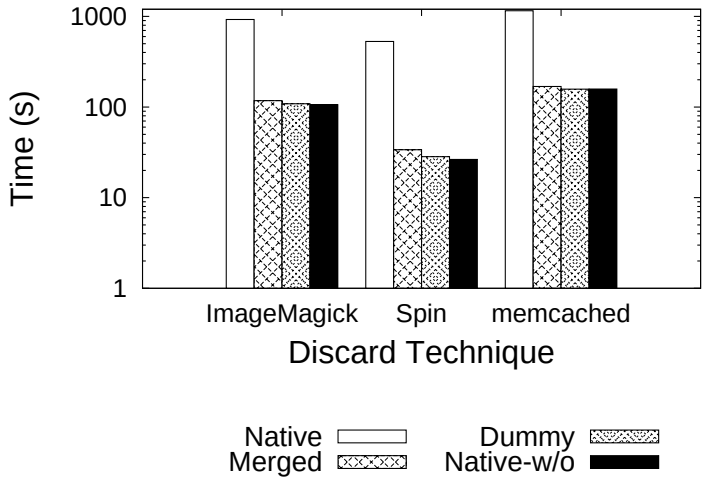


Figure 3.7: **G**arbage Collection Performance: Impact of merged and dummy discards on application performance for FlashVM. y-axis is log-scale for application execution time.

Figure 3.7 compares the performance of four different systems: FlashVM with merged discards over 100 MB ranges, FlashVM with dummy discards, native Linux VM with discard support and baseline Linux VM without discard support. Linux with discard is 12 times slower than the baseline system, indicating the high cost for inefficient use of the discard command. In contrast, FlashVM with merged discard, which also has the reliability benefits of Linux with discard, is only 15 *percent* slower than baseline. With the addition of adaptive dummy discards, which reduces the rate of discards when page clusters are rapidly allocated,

Workload	**D**iskVM		**F**lashVM	
	Runtime	Mem	Const Mem Rel. Runtime	Const Runtime Rel. Memory
ImageMagick	207	814	31%	51%
Spin	209	795	11%	16%
SpecJBB	275	710	6%	19%
memcached-store	396	706	18%	60%
memcached-lookup	257	837	23%	50%

Table 3.3: **C**ost/Benefit Analysis: FlashVM analysis for different memory-intensive application workloads. Systems compared are DiskVM with disk-backed VM, a FlashVM system with the same memory (Const Mem) and one with the same performance but less memory (Const Runtime). FlashVM results show the execution time and memory usage, both relative to DiskVM. Application execution time Runtime is in seconds; memory usage Mem is in megabytes.

performance is 11% slower than baseline. In all cases, the slowdown is due to the long latency of discard operations, which have little direct performance benefit. These results demonstrate that naive use of discard greatly degrades performance, while FlashVM's merged and dummy discard achieve similar reliability benefits at performance near native speeds.

3.4.4 FlashVM Application Performance

Adoption of FlashVM is fundamentally an economic decision: a FlashVM system can perform better than a DiskVM system even when it is provisioned with more expensive DRAM. Therefore, we evaluate the performance gains and memory savings when replacing disk with flash for paging. Our results reflect estimates for absolute memory savings in megabytes.

Table 3.3 presents the performance and memory usage of five application workloads on three systems:

1. *DiskVM* with 1 GB memory and a dedicated disk for swapping;
2. *FlashVM - Const Mem* with the same DRAM size as DiskVM, but improved performance;
3. *FlashVM - Const Runtime* with reduced DRAM size, but same performance as DiskVM.

Our analysis in Table 3.3 represents configurations that correspond to the three data points shown in Figure 3.1. System configurations for workloads with high locality or unused memory do not page and show no benefit from FlashVM. Similarly, those with no locality or extreme memory requirements lie on the far left in Figure 3.1 and perform so poorly as to be unusable. Such data points are not useful for analyzing virtual memory performance. The column in Table 3.3 titled DiskVM shows the execution time and memory usage of the five workloads on a system swapping to disk. Under FlashVM - Const Mem and FlashVM - Const Runtime, we show the percentage reduction in the execution time and memory usage respectively, both when compared to DiskVM. The reduction in memory usage corresponds to the potential price savings by swapping to flash rather than disk for achieving similar performance.

For all applications, a FlashVM system outperforms a system configured with the same amount of DRAM and disk-backed VM (FlashVM - Const Mem against DiskVM). FlashVM's reduction in execution time varies from 69% for ImageMagick to 94% for the modified SpecJBB, a 3-16x speedup. On average, FlashVM reduces run time by 82% over DiskVM. Similarly, we find that there is a potential 60% reduction in the amount of DRAM required on the FlashVM system to achieve similar performance as DiskVM (FlashVM - Const Runtime against DiskVM). This benefit comes directly from the lower access latency and higher bandwidth of flash, and results in both price and power savings for the FlashVM system.

Overall, we find that applications with poor locality have higher memory savings because the memory saved does not substantially increase their page fault rate. In contrast, applications with good locality see proportionally more page faults from each lost memory page. Furthermore, applications also benefit differently depending on their access patterns. For example, when storing objects, *memcached* server performance improves 5x on a FlashVM system with the same memory size, but only 4.3x for a lookup workload. The memory savings differ similarly.

3.5 Related Work

The FlashVM design draws on past work investigating the use of solid-state memory for storage. We categorize this work into the following four classes:

Persistent Storage. Flash has most commonly been proposed as a storage system to replace disks. eNVy presented a storage system that placed flash on the memory bus with a special controller equipped with a battery-backed SRAM buffer [114]. File systems, such as YAFFS and JFFS2 [93], manage flash to hide block erase latencies and perform wear-leveling to handle bad blocks. More recently, TxFlash exposes a novel transactional interface to use flash memory by exploiting its copy-on-write nature [91]. These systems all treat flash as persistent storage, similar to a file system. In contrast, FlashVM largely ignores the non-volatile aspect of flash and instead focuses on the design of a high-performance, reliable and scalable virtual memory.

Hybrid Systems. Guided by the price and performance of flash, hybrid systems propose flash as a second-level cache between memory and disk. FlashCache uses flash as secondary file/buffer cache to provide a larger caching tier than DRAM [51]. Windows and Solaris can use USB flash drives and solid-state disks as read-optimized disk caches managed by the file system [8, 35]. All these systems treat flash as a cache of the contents on a disk and mainly exploit its performance benefits. In contrast, FlashVM treats flash as a backing store for evicted pages, accelerates both read and write operations, and provides mechanisms for improving flash reliability and efficiency of garbage collection by using the semantic information about paging only available within the OS.

Non-volatile Memory. NAND flash is the only memory technology after DRAM that has become cheap and ubiquitous in the last few decades. Other non-volatile storage class memory technologies like phase-change memory (PCM) and magneto-resistive memory (MRAM) are expected to come at par with DRAM prices by 2015 [79]. Recent proposals have advocated the use of PCM as a first-level memory placed on the memory bus alongside DRAM [55, 72]. In contrast, FlashVM adopts cheap NAND flash and incorporates it as swap space rather than memory directly addressable by user-mode programs.

Virtual Memory. Past proposals on using flash as virtual memory focused on new page-replacement schemes [87] or providing compiler-assisted, energy-efficient swap space for embedded systems [60, 86]. In contrast, FlashVM seeks more OS control for memory management on flash, while addressing three major problems for paging to dedicated flash. Further, we present the first description of the usage of the discard command on a real flash device and provide mechanisms to optimize the performance of garbage collection.

3.6 Summary

FlashVM adapts the Linux virtual memory system for the performance, reliability, and garbage collection characteristics of flash storage. In examining Linux, we find many dependencies on the performance characteristics of disks, as in the case of prefetching only adjacent pages. While the assumptions about disk performance are not made explicit, they permeate the design, particularly regarding batching of requests to reduce seek latencies and to amortize the cost of I/O. As new storage technologies with yet different performance characteristics and challenges become available, such as memristors and phase-change memory, it will be important to revisit both operating system and application designs. Over the last two years, several systems using flash SSDs as an extension to memory similar to FlashVM have emerged as popular industry products [32, 30, 109].

4 FLASHTIER: FAST AND CONSISTENT STORAGE CACHE

Solid-state drives (SSDs) composed of multiple flash memory chips are often deployed as a cache in front of cheap and slow disks [51, 27]. This provides the performance of flash with the cost of disk for large data sets, and is actively used by Facebook and others to provide low-latency access to petabytes of data [28, 104, 96, 84]. Many vendors sell dedicated caching products that pair an SSD with proprietary software that runs in the OS to migrate data between the SSD and disks [48, 80, 29] to improve storage performance.

Building a cache upon a standard SSD, though, is hindered by the narrow block interface and internal block management of SSDs, which are designed to serve as a disk replacement [5, 91, 114]. Caches have at least three different behaviors that distinguish them from general-purpose storage. First, data in a cache may be present elsewhere in the system, and hence need not be durable. Thus, caches have more flexibility in how they manage data than a device dedicated to storing data persistently. Second, a cache stores data from a separate address space, the disks', rather than at native addresses. Thus, using a standard SSD as a cache requires an additional step to map block addresses from the disk into SSD addresses for the cache. If the cache has to survive crashes, this map must be persistent. Third, the consistency requirements for caches differ from storage devices. A cache must ensure it never returns stale data, but can also return nothing if the data is not present. In contrast, a storage device provides ordering guarantees on when writes become durable.

This chapter describes *FlashTier*, a system that explores the opportunities for tightly integrating solid-state caching devices into the storage hierarchy. First, we investigate how small changes to the interface and internal block management of conventional SSDs can result in a much more effective caching device, a *solid-state cache*. Second, we investigate how such a dedicated caching device changes *cache managers*, the software component responsible for migrating data between the flash caching tier and disk storage. This design provides a clean separation between the caching device and its internal structures, the system software managing the cache, and the disks storing data.

The remainder of the chapter is structured as follows. Section 4.1 describes our caching

workload characteristics and motivates FlashTier. Section 4.2 presents an overview of FlashTier design, followed by a detailed description in Section 4.3 and 4.4. We evaluate FlashTier design techniques in Section 4.5.

4.1 Motivation

Flash is an attractive technology for caching because its price and performance are between DRAM and disk: about five times cheaper than DRAM and an order of magnitude (or more) faster than disk (see Table 2.1). Furthermore, its persistence enables cache contents to survive crashes or power failures, and hence can improve cold-start performance. As a result, SSD-backed caching is popular in many environments including workstations, virtualized enterprise servers, database backends, and network disk storage [80, 76, 96, 98, 51].

Flash has two characteristics that require special management to achieve high reliability and performance. First, flash does not support *in-place writes*. Instead, a block of flash must be *erased* (a lengthy operation) before it can be written. Second, to support writing a block multiple times, flash devices use *address mapping* to translate block addresses received from a host into physical locations in flash. This mapping allows a block to be written out-of-place to a pre-erased block rather than erasing and rewriting in-place. As a result, SSDs employ *garbage collection* to compact data and provide free, erased blocks for upcoming writes.

The motivation for FlashTier is the observation that caching and storage have different behavior and different requirements. We next study three aspects of caching behavior to distinguish it from general-purpose storage. Our study uses traces from two different sets of production systems downstream of an active page cache over 1-3 week periods [53, 74]. These systems have different I/O workloads that consist of a file server (*homes* workload), an email server (*mail* workload) and file servers from a small enterprise data center hosting user home and project directories (*usr* and *proj*). Table 4.4 summarizes the workload statistics. Trends observed across all these workloads directly motivate our design for FlashTier.

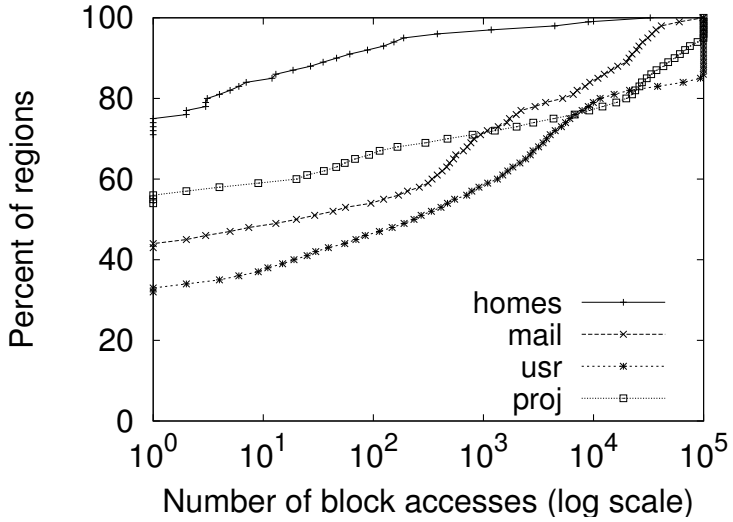


Figure 4.1: **L**ogical Block Addresses Distribution: The distribution of unique block accesses across 100,000 4 KB block regions of the disk address space.

Address Space Density. A hard disk or SSD exposes an address space of the same size as its capacity. As a result, a mostly full disk will have a dense address space, because there is valid data at most addresses. In contrast, a cache stores only *hot data* that is currently in use. Thus, out of the terabytes of storage, a cache may only contain a few gigabytes. However, that data may be at addresses that range over the full set of possible disk addresses.

Figure 4.1 shows the density of requests to 100,000-block regions of the disk address space. To emulate the effect of caching, we use only the top 25% most-accessed blocks from each trace (those likely to be cached). Across all four traces, more than 55% of the regions get less than 1% of their blocks referenced, and only 25% of the regions get more than 10%. These results motivate a change in how mapping information is stored within an SSC as compared to an SSD: while an SSD should optimize for a *dense address space*, where most addresses contain data, an SSC storing only active data should instead optimize for a *sparse address space*.

Persistence and Cache Consistency. Disk caches are most effective when they offload workloads that perform poorly, such as random reads and writes. However, large caches and poor disk performance for such workloads result in exceedingly long cache warming periods.

For example, filling a 100 GB cache from a 500 IOPS disk system takes over 14 hours. Thus, caching data persistently across system restarts can greatly improve cache effectiveness.

On an SSD-backed cache, maintaining cached data persistently requires storing cache metadata, such as the state of every cached block and the mapping of disk blocks to flash blocks. On a clean shutdown, this can be written back at low cost. However, to make cached data *durable* so that it can survive crash failure, is much more expensive. Cache metadata must be persisted on every update, for example when updating or invalidating a block in the cache. These writes degrade performance, and hence many caches do not provide crash recovery [14, 35], and discard all cached data after a crash.

A hard disk or SSD provides crash recovery with simple consistency guarantees to the operating system: barriers ensure that preceding requests complete before subsequent ones are initiated. For example, a barrier can ensure that a journal commit record only reaches disk *after* the journal entries [20]. However, barriers provide ordering between requests to a single device, and do not address consistency between data on different devices. For example, a write sent both to a disk and a cache may complete on just one of the two devices, but the combined system must remain consistent.

Thus, the guarantee a cache makes is semantically different than ordering: a cache should never return stale data, and should never lose dirty data. However, within this guarantee, the cache has freedom to relax other guarantees, such as the persistence of clean data.

Wear Management. A major challenge with using SSDs as a disk cache is their limited write endurance: a single MLC flash cell can only be erased 10,000 times. In addition, garbage collection is often a contributor to wear, as live data must be copied to make free blocks available. A recent study showed that more than 70% of the erasures on a full SSD were due to garbage collection [24].

Furthermore, caching workloads are often more intensive than regular storage workloads: a cache stores a greater fraction of hot blocks, which are written frequently, as compared to a general storage workload. In looking at the top 25% most frequently referenced blocks in two write-intensive storage traces, we find that the average writes per block is 4 times greater than

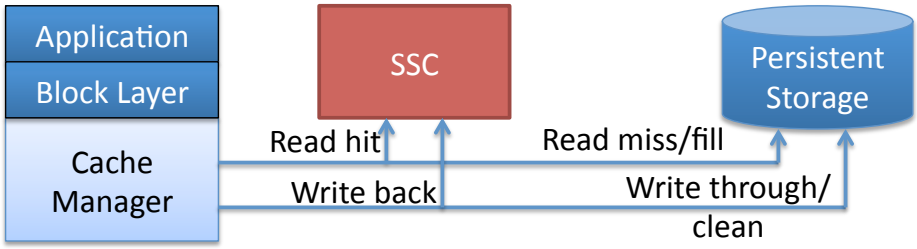


Figure 4.2: **F**lashTier Data Path: A cache manager forwards block read/write requests to disk and solid-state cache.

for the trace as a whole, indicating that caching workloads are likely to place greater durability demands on the device. Second, caches operate at full capacity while storage devices tend to be partially filled. At full capacity, there is more demand for garbage collection. This can hurt reliability by copying data more frequently to make empty blocks [46, 3].

4.2 Design Overview

FlashTier is a block-level caching system, suitable for use below a file system, virtual memory manager, or database. A *cache manager* interposes above the disk device driver in the operating system to send requests to the either the flash device or the disk, while a *solid-state cache (SSC)* stores and assists in managing cached data. Figure 4.2 shows the flow of read and write requests from the application to SSC and disk-storage tiers from the cache manager.

4.2.1 Cache Management

The cache manager receives requests from the block layer and decides whether to consult the cache on reads, and whether to cache data on writes. On a cache miss, the manager sends the request to the disk tier, and it may optionally store the returned data in the SSC.

FlashTier supports two modes of usage: *write-through* and *write-back*. In write-through

mode, the cache manager writes data to the disk and populates the cache either on read requests or at the same time as writing to disk. In this mode, the SSC contains only clean data, and is best for read-heavy workloads, where there is little benefit to caching writes, and when the cache is not considered reliable, as in a client-side cache for networked storage. In this mode, the cache manager consults the SSC on every read request. If the data is not present, the SSC returns an error, and the cache manager fetches the data from disk. On a write, the cache manager must either evict the old data from the SSC or write the new data to it.

In write-back mode, the cache manager may write to the SSC without updating the disk. Thus, the cache may contain dirty data that is later evicted by writing it back to the disk. This complicates cache management, but performs better with write-heavy workloads and local disks. In this mode, the cache manager must actively manage the contents of the cache to ensure there is space for new data. The cache manager maintains a table of dirty cached blocks to track which data is in the cache and ensure there is enough space in the cache for incoming writes. The manager has two options to make free space: it can evict a block, which guarantees that subsequent reads to the block will fail, and allows the manager to direct future reads of the block to disk. Or, the manager notifies the SSC that the block is clean, which then allows the SSC to evict the block in the future. In the latter case, the manager can still consult the cache on reads and must evict/overwrite the block on writes if it still exists in the cache.

4.2.2 Addressing

With an SSD-backed cache, the manager must maintain a *mapping table* to store the block's location on the SSD. The table is indexed by logical block number (LBN), and can be used to quickly test whether block is in the cache. In addition, the manager must track free space and evict data from the SSD when additional space is needed. It does this by removing the old mapping from the mapping table, inserting a mapping for a new LBN with the same SSD address, and then writing the new data to the SSD.

In contrast, an SSC does not have its own set of addresses. Instead, it exposes a *unified address space*: the cache manager can write to an SSC using logical block numbers (or disk addresses), and the SSC internally maps those addresses to physical locations in flash. As flash devices already maintain a mapping table to support garbage collection, this change does not introduce new overheads. Thus the cache manager in FlashTier no longer needs to store the mapping table persistently, because this functionality is provided by the SSC.

The large address space raises the new possibility that cache does not have capacity to store the data, which means the cache manager must ensure not to write too much data or the SSC must evict data to make space.

4.2.3 Space Management

As a cache is much smaller than the disks that it caches, it requires mechanisms and policies to manage its contents. For write-through caching, the data is clean, so the SSC may silently evict data to make space. With write-back caching, though, there may be a mix of clean and dirty data in the SSC. An SSC exposes three mechanisms to cache managers for managing the cached data: *evict*, which forces out a block; *clean*, which indicates the data is clean and can be evicted by the SSC, and *exists*, which tests for the presence of a block and is used during recovery. As described above, for write-through caching all data is clean, whereas with write-back caching, the cache manager must explicitly clean blocks after writing them back to disk.

The ability to evict data can greatly simplify space management within the SSC. Flash drives use garbage collection to compact data and create freshly erased blocks to receive new writes, and may relocate data to perform wear leveling, which ensures that erases are spread evenly across the physical flash cells. This has two costs. First, copying data for garbage collection or for wear leveling reduces performance, as creating a single free block may require reading and writing multiple blocks for compaction. Second, an SSD may copy and compact data that is never referenced again. An SSC, in contrast, can evict data rather than copying it. This speeds garbage collection, which can now erase clean blocks without copying their live

data because clean cache blocks are also available in disk. If the data is not later referenced, this has little impact on performance. If the data is referenced later, then it must be re-fetched from disk and cached again.

Finally, a cache does not require overprovisioned blocks to make free space available. Most SSDs reserve 5-20% of their capacity to create free erased blocks to accept writes. However, because an SSC does not promise a fixed capacity, it can flexibly dedicate space either to data, to reduce miss rates, or to the log, to accept writes.

4.2.4 Crash Behavior

Flash storage is persistent, and in many cases it would be beneficial to retain data across system crashes. For large caches in particular, a durable cache can avoid an extended warm-up period where all data must be fetched from disks. However, to be usable after a crash, the cache must retain the metadata mapping disk blocks to flash blocks, and must guarantee correctness by never returning stale data. This can be slow, as it requires synchronous metadata writes when modifying the cache. As a result, many SSD-backed caches, such as Solaris L2ARC and NetApp Mercury, must be reset after a crash [14, 35].

The challenge in surviving crashes in an SSD-backed cache is that the mapping must be persisted along with cached data, and the consistency between the two must also be guaranteed. This can greatly slow cache updates, as replacing a block requires writes to: (i) remove the old block from the mapping, (ii) write the new data, and (iii) add the new data to the mapping.

4.2.5 Guarantees

FlashTier provides consistency and durability guarantees over cached data in order to allow caches to survive a system crash. The system distinguishes *dirty data*, for which the newest copy of the data may only be present in the cache, from *clean data*, for which the underlying disk also has the latest value.

1. A read following a write of dirty data will return that data.

2. A read following a write of clean data will return *either* that data or a not-present error.

3. A read following an eviction will return a not-present error.

The first guarantee ensures that dirty data is durable and will not be lost in a crash. The second guarantee ensures that it is *always* safe for the cache manager to consult the cache for data, as it must either return the newest copy or an error. Finally, the last guarantee ensures that the cache manager can invalidate data in the cache and force subsequent requests to consult the disk. Implementing these guarantees within the SSC is much simpler than providing them in the cache manager, as a flash device can use internal transaction mechanisms to make all three writes at once [91, 85].

4.3 System Design

FlashTier has three design goals to address the limitations of caching on SSDs:

- *Address space management* to unify address space translation and block state between the OS and SSC, and optimize for sparseness of cached blocks.
- *Free space management* to improve cache write performance by silently evicting data rather than copying it within the SSC.
- *Consistent interface* to provide consistent reads after cache writes and eviction, and make both clean and dirty data as well as the address mapping durable across a system crash or reboot.

This section discusses the design of FlashTier's address space management, block interface and consistency guarantees of SSC, and free space management.

4.3.1 Unified Address Space

FlashTier unifies the address space and cache block state split between the cache manager running on host and firmware in SSC. Unlike past work on virtual addressing in SSDs [50], the address space in an SSC may be very sparse because caching occurs at the block level.

Sparse Mapping. The SSC optimizes for sparseness in the blocks it caches with a *sparse hash map* data structure, developed at Google [34]. This structure provides high performance and low space overhead for sparse hash keys. In contrast to the mapping structure used by Facebook's FlashCache, it is fully associative and thus must encode the complete block address for lookups.

The map is a hash table with $\mathtt{t}$ buckets divided into $\mathtt{t}/\mathsf{M}$ groups of M buckets each. Each group is stored sparsely as an array that holds values for allocated block addresses and an occupancy bitmap of size M, with one bit for each bucket. A bit at location $\mathtt{i}$ is set to 1 if and only if bucket $\mathtt{i}$ is non-empty. A lookup for bucket $\mathtt{i}$ calculates the value location from the number of 1s in the bitmap before location $\mathtt{i}$. We set M to 32 buckets per group, which reduces the overhead of bitmap to just 3.5 bits per key, or approximately 8.4 bytes per occupied entry for 64-bit memory pointers [34]. The runtime of all operations on the hash map is bounded by the constant M, and typically there are no more than 4-5 probes per lookup.

The SSC keeps the entire mapping in its memory. However, the SSC maps a fixed portion of the flash blocks at a 4 KB page granularity and the rest at the granularity of an 256 KB erase block, similar to hybrid FTL mapping mechanisms [58, 46]. The mapping data structure supports lookup, insert and remove operations for a given key-value pair. Lookups return the physical flash page number for the logical block address in a request. The physical page number addresses the internal hierarchy of the SSC arranged as flash package, die, plane, block and page. Inserts either add a new entry or overwrite an existing entry in the hash map. For a remove operation, an invalid or unallocated bucket results in reclaiming memory and the occupancy bitmap is updated accordingly. Therefore, the size of the sparse hash map grows with the actual number of entries, unlike a linear table indexed by a logical or physical address.

Block State. In addition to the logical-to-physical map, the SSC maintains additional data for internal operations, such as the state of all flash blocks for garbage collection and usage statistics to guide wear-leveling and eviction policies. This information is accessed by physical

Name	**F**unction
write-dirty	Insert new block or update existing block with dirty data.
write-clean	Insert new block or update existing block with clean data.
read	Read block if present or return error.
evict	Evict block immediately.
clean	Allow future eviction of block.
exists	Test for presence of dirty blocks.

Table 4.1: **T**he Solid-State Cache Interface.

address only, and therefore can be stored in the out-of-band (OOB) area of each flash page. This is a small area (64–224 bytes) associated with each page [17] that can be written at the same time as data. To support fast address translation for physical addresses when garbage collecting or evicting data, the SSC also maintains a reverse map, stored in the OOB area of each page and updates it on writes. With each block-level map entry in device memory, the SSC also stores a dirty-block bitmap recording which pages within the erase block contain dirty data.

4.3.2 Consistent Cache Interface

FlashTier provides a consistent cache interface that reflects the needs of a cache to (i) persist cached data across a system reboot or crash, and (ii) never return stale data because of an inconsistent mapping. Most SSDs provide the read/write interface of disks, augmented with a *trim* command to inform the SSD that data need not be saved during garbage collection. However, the existing interface is insufficient for SSD caches because it leaves undefined what data is returned when reading an address that has been written or evicted [75]. An SSC, in contrast, provides an interface with precise guarantees over consistency of both cached data and mapping information. The SSC interface is a small extension to the standard SATA/SCSI read/write/trim commands.

Interface

FlashTier's interface consists of six operations, as listed in Table 4.1. We next describe these operations and their usage by the cache manager in more detail.

Writes. FlashTier provides two write commands to support write-through and write-back caching. For write-back caching, the *write-dirty* operation guarantees that data is durable before returning. This command is similar to a standard SSD write, and causes the SSC also update the mapping, set the dirty bit on the block and save the mapping to flash using logging. The operation returns only when the data and mapping are durable in order to provide a consistency guarantee.

The *write-clean* command writes data and marks the block as clean, so it can be evicted if space is needed. This operation is intended for write-through caching and when fetching a block into the cache on a miss. The guarantee of *write-clean* is that a subsequent read will return either the new data or a not-present error, and hence the SSC must ensure that data and metadata writes are properly ordered. Unlike *write-dirty*, this operation can be buffered; if the power fails before the write is durable, the effect is the same as if the SSC silently evicted the data. However, if the write replaces previous data at the same address, the mapping change must be durable before the SSC completes the request.

Reads. A read operation looks up the requested block in the device map. If it is present it returns the data, and otherwise returns an error. The ability to return errors from reads serves three purposes. First, it allows the cache manager to request any block, without knowing if it is cached. This means that the manager need not track the state of all cached blocks precisely; approximation structures such as a Bloom Filter can be used safely to prevent reads that miss in the SSC. Second, it allows the SSC to manage space internally by evicting data. Subsequent reads of evicted data return an error. Finally, it simplifies the consistency guarantee: after a block is written with *write-clean*, the cache can still return an error on reads. This may occur if a crash occurred after the write but before the data reached flash.

Eviction. FlashTier also provides a new *evict* interface to provide a well-defined read-after-evict semantics. After issuing this request, the cache manager knows that the cache cannot contain the block, and hence is free to write updated versions to disk. As part of the eviction, the SSC removes the forward and reverse mappings for the logical and physical pages from the hash maps and increments the number of invalid pages in the erase block. The durability guarantee of *evict* is similar to *write-dirty*: the SSC ensures the eviction is durable before completing the request.

Explicit eviction is used to invalidate cached data when writes are sent only to the disk. In addition, it allows the cache manager to precisely control the contents of the SSC. The cache manager can leave data dirty and explicitly evict selected victim blocks. Our implementation, however, does not use this policy.

Block cleaning. A cache manager indicates that a block is clean and may be evicted with the *clean* command. It updates the block metadata to indicate that the contents are clean, but does not touch the data or mapping. The operation is asynchronous, after a crash cleaned blocks may return to their dirty state.

A write-back cache manager can use *clean* to manage the capacity of the cache: the manager can clean blocks that are unlikely to be accessed to make space for new writes. However, until the space is actually needed, the data remains cached and can still be accessed. This is similar to the management of free pages by operating systems, where page contents remain usable until they are rewritten.

Testing with *exists*. The *exists* operation allows the the cache manager to query the state of a range of cached blocks. The cache manager passes a block range, and the SSC returns the dirty bitmaps from mappings within the range. As this information is stored in the SSC's memory, the operation does not have to scan flash. The returned data includes a single bit for each block in the requested range that, if set, indicates the block is present and dirty. If the block is not present or clean, the bit is cleared. While this version of *exists* returns only dirty blocks, it could be extended to return additional per-block metadata, such as access

time or frequency, to help manage cache contents.

This operation is used by the cache manager for recovering the list of dirty blocks after a crash. It scans the entire disk address space to learn which blocks are dirty in the cache so it can later write them back to disk.

Persistence

SSCs rely on a combination of logging, checkpoints, and out-of-band writes to persist its internal data. Logging allows low-latency writes to data distributed throughout memory, while checkpointing provides fast recovery times by keeping the log short. Out-of-band writes provide a low-latency means to write metadata near its associated data. However, out-of-band area may be a scarce resource shared for different purposes, for example, storing other metadata or error-correction codes, as we find later while prototyping the SSC design on the OpenSSD Jasmine board [107].

Logging. An SSC uses an operation log to persist changes to the sparse hash map. A log record consists of a monotonically increasing log sequence number, the logical and physical block addresses, and an identifier indicating whether this is a page-level or block-level mapping.

For operations that may be buffered, such as *clean* and *write-clean*, an SSC uses asynchronous group commit [40] to flush the log records from device memory to flash device periodically. For operations with immediate consistency guarantees, such as *write-dirty* and *evict*, the log is flushed as part of the operation using a synchronous commit. For example, when updating a block with *write-dirty*, the SSC will create a log record invalidating the old mapping of block number to physical flash address and a log record inserting the new mapping for the new block address. These are flushed using an atomic-write primitive [85] to ensure that transient states exposing stale or invalid data are not possible.

In the absence of hardware support for an atomic-write primitive and out-of-band area access, while prototyping the SSC design on the OpenSSD board [107] (see Chapter 5), we use the last page in each erase block to log mapping updates.

Checkpointing. To ensure faster recovery and small log size, SSCs checkpoint the mapping data structure periodically so that the log size is less than a fixed fraction of the size of checkpoint. This limits the cost of checkpoints, while ensuring logs do not grow too long. It only checkpoints the forward mappings because of the high degree of sparseness in the logical address space. The reverse map used for invalidation operations and the free list of blocks are clustered on flash and written in-place using out-of-band updates to individual flash pages. FlashTier maintains two checkpoints on dedicated regions spread across different planes of the SSC that bypass address translation.

For prototoyping SSC on OpenSSD platform, we defer checkpoints until requested by host using a *flush* barrier operation, which allows OS and application define consistent points, and minimizes interference between normal I/O and checkpoint write traffic.

Recovery. The recovery operation reconstructs the different mappings in device memory after a power failure or reboot. It first computes the difference between the sequence number of the most recent committed log record and the log sequence number corresponding to the beginning of the most recent checkpoint. It then loads the mapping checkpoint and replays the log records falling in the range of the computed difference. The SSC performs roll-forward recovery for both the page-level and block-level maps, and reconstructs the reverse-mapping table from the forward tables.

4.3.3 Free Space Management

FlashTier provides high write performance by leveraging the semantics of caches for garbage collection. SSDs use garbage collection to compact data and create free erased blocks. Internally, flash is organized as a set of *erase blocks*, which contain a number of pages, typically 64. Garbage collection coalesces the live data from multiple blocks and erases blocks that have no valid data. If garbage collection is performed frequently, it can lead to *write amplification*, where data written once must be copied multiple times, which hurts performance and reduces the lifetime of the drive [46, 3].

The hybrid flash translation layer in modern SSDs separates the drive into data blocks and log blocks. New data is written to the log and then merged into data blocks with garbage collection. The data blocks are managed with block-level translations (256 KB) while the log blocks use finer-grained 4 KB translations. Any update to a data block is performed by first writing to a log block, and later doing a *full merge* that creates a new data block by merging the old data block with the log blocks containing overwrites to the data block.

Silent eviction. SSCs leverage the behavior of caches by evicting data when possible rather than copying it as part of garbage collection. FlashTier implements a *silent eviction* mechanism by integrating cache replacement with garbage collection. The garbage collector selects a flash plane to clean and then selects the top-k victim blocks based on a policy described below. It then removes the mappings for any valid pages within the victim blocks, and erases the victim blocks. Unlike garbage collection, FlashTier does not incur any copy overhead for rewriting the valid pages.

When using silent eviction, an SSC will only consider blocks written with *write-clean* or explicitly cleaned. If there are not enough candidate blocks to provide free space, it reverts to regular garbage collection. Neither *evict* nor *clean* operations trigger silent eviction; they instead update metadata indicating a block is a candidate for eviction during the next collection cycle.

Policies. We have implemented two policies to select victim blocks for eviction. Both policies only apply silent eviction to data blocks and use a cost-benefit based mechanism to select blocks for eviction. Cost is defined as the number of clean valid pages (*i.e.*, utilization) in the erase block selected for eviction. Benefit is defined as the age (last modified time) for the erase block. Both policies evict an erase block with the minimum cost/benefit ratio, similar to segment cleaning in log-structured file systems [95].

The two policies differ in whether a data block is converted into a log or data block after silent eviction. The first policy used in the *SSC* device only creates erased *data blocks* and not log blocks, which still must use normal garbage collection. The second policy used in the

SSC-V device uses the same cost/benefit policy for selecting candidate victims, but allows the erased blocks to be used for either data or logging. This allows a *v*ariable number of log blocks, which reduces garbage collection costs: with more log blocks, garbage collection of them is less frequent, and there may be fewer valid pages in each log block. However, this approach increases memory usage to store fine-grained translations for each block in the log. With *SSC-V*, new data blocks are created via switch merges, which convert a sequentially written log block into a data block without copying data.

4.3.4 Cache Manager

The cache manager is based on Facebook's FlashCache for Linux [28]. It provides support for both write-back and write-through caching modes and implements a recovery mechanism to enable cache use after a crash.

The write-through policy consults the cache on every read. As read misses require only access to the in-memory mapping, these incur little delay. The cache manager, fetches the data from the disk on a miss and writes it to the SSC with *write-clean*. Similarly, the cache manager sends new data from writes both to the disk and to the SSC with *write-clean*. As all data is clean, the manager never sends any *clean* requests. We optimize the design for memory consumption assuming a high hit rate: the manager stores no data about cached blocks, and consults the cache on every request. An alternative design would be to store more information about which blocks are cached in order to avoid the SSC on most cache misses.

The write-back mode differs on the write path and in cache management; reads are handled similarly to write-through caching. On a write, the cache manager use *write-dirty* to write the data to the SSC only. The cache manager maintains an in-memory table of cached dirty blocks. Using its table, the manager can detect when the percentage of dirty blocks within the SSC exceeds a set threshold, and if so issues *clean* commands for LRU blocks. Within the set of LRU blocks, the cache manager prioritizes cleaning of contiguous dirty blocks, which can be merged together for writing to disk. The cache manager then removes the state of the clean block from its table.

The dirty-block table is stored as a linear hash table containing metadata about each dirty block. The metadata consists of an 8-byte associated disk block number, an optional 8-byte checksum, two 2-byte indexes to the previous and next blocks in the LRU cache replacement list, and a 2-byte block state, for a total of 14-22 bytes.

After a failure, a write-through cache manager may immediately begin using the SSC. It maintains no transient in-memory state, and the cache-consistency guarantees ensure it is safe to use all data in the SSC. Similarly, a write-back cache manager can also start using the cache immediately, but must eventually repopulate the dirty-block table in order to manage cache space. The cache manager scans the entire disk address space with *exists*. This operation can overlap normal activity and thus does not delay recovery.

4.4 Implementation

The implementation of FlashTier entails three components: the cache manager, an SSC functional emulator, and an SSC timing simulator. The first two are Linux kernel modules (kernel 2.6.33), and the simulator models the time for the completion of each request.

We base the cache manager on Facebook's FlashCache [28]. We modify its code to implement the cache policies described in the previous section. In addition, we added a trace-replay framework invokable from user-space with direct I/O calls to evaluate performance.

4.4.1 SSC Simulator Implementation

We base the SSC simulator on FlashSim [52]. The simulator provides support for an SSD controller, and a hierarchy of NAND-flash packages, planes, dies, blocks and pages. We enhance the simulator to support page-level and hybrid mapping with different mapping data structures for address translation and block state, write-ahead logging with synchronous and asynchronous group commit support for insert and remove operations on mapping, periodic checkpointing from device memory to a dedicated flash region, and a roll-forward recovery logic to reconstruct the mapping and block state. We have two basic configurations of the simulator, targeting the two silent eviction policies. The first configuration (termed *SSC*

in the evaluation) uses the first policy and statically reserves a portion of the flash for log blocks and provisions enough memory to map these with page-level mappings. The second configuration, *SSC-V*, uses the second policy and allows the fraction of log blocks to vary based on workload but must reserve memory capacity for the maximum fraction at page level. In our tests, we fix log blocks at 7% of capacity for SSC and allow the fraction to range from 0-20% for SSC-V.

We implemented our own FTL that is similar to the FAST FTL [58]. We integrate silent eviction with background and foreground garbage collection for data blocks, and with merge operations for *SSC-V* when recycling log blocks [3]. We also implement inter-plane copy of valid pages for garbage collection (where pages collected from one plane are written to another) to balance the number of free blocks across all planes. The simulator also tracks the utilization of each block for the silent eviction policies.

The SSC emulator is implemented as a block device and uses the same code for SSC logic as the simulator. In order to emulate large caches efficiently (much larger than DRAM), it stores the metadata of all cached blocks in memory but discards data on writes and returns fake data on reads, similar to David [4].

4.4.2 SSC Prototype Implementation

We use the OpenSSD platform [107] (see Figure 4.3) to prototype the SSC design. It is the most up-to-date open platform available today for prototyping new SSD designs. It uses a commercial flash controller for managing flash at speeds close to commodity SSDs. We prototype FlashTier SSC to verify its practicality and validate if it performs as we projected in simulation earlier.

OpenSSD Research Platform The OpenSSD board is designed as a platform for implementing and evaluating SSD firmware and is sponsored primarily by Indilinx, an SSD-controller manufacturer [107]. The board is composed of commodity SSD parts: an Indilinx Barefoot ARM-based SATA controller, introduced in 2009 for second generation SSDs and still used in many commercial SSDs; 96 KB SRAM; 64 MB DRAM for storing the flash translation

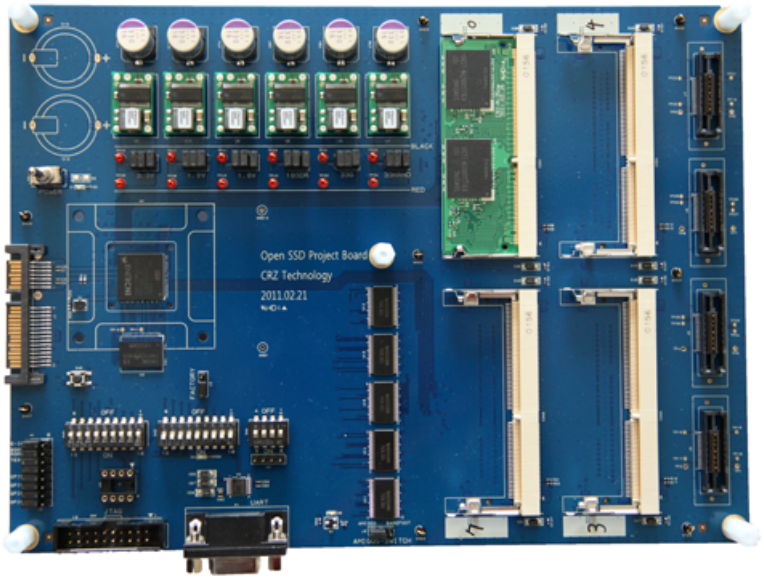


Figure 4.3: **O**penSSD Architecture: Major components of OpenSSD platform are the Indilinx Barefoot SSD controller, internal SRAM, SDRAM, NAND flash, specialized hardware for buffer management, flash control, and memory utility functions; and debugging UART/JTAG ports.

Controller	ARM7TDMI-S	Frequency	87.5 MHz
SDRAM	64 MB (4 B ECC/128 B)	Frequency	175 MHz
Flash	256 GB	Overprovisioning	7%
Type	MLC async mode	Packages	4
Dies/package	2	Banks/package	4
Channel Width	2 bytes	Ways	2
Physical Page	8 KB (448 B spare)	Physical Block	2 MB
Virtual Page	32 KB	Virtual Block	4 MB

Table 4.2: **O**penSSD device configuration.

mapping and for SATA buffers; and 8 slots holding up to 256 GB of MLC NAND flash. The controller runs firmware that can send read/write/erase and copyback (copy data within a bank) operations to the flash banks over a 16-bit I/O channel. The chips use two planes and have 8 KB physical pages. The device uses large 32 KB virtual pages, which improve performance by striping data across physical pages on 2 planes on 2 chips within a flash bank. Erase blocks are 4 MB and composed of 128 contiguous virtual pages.

The controller provides hardware support to accelerate command processing in the form of

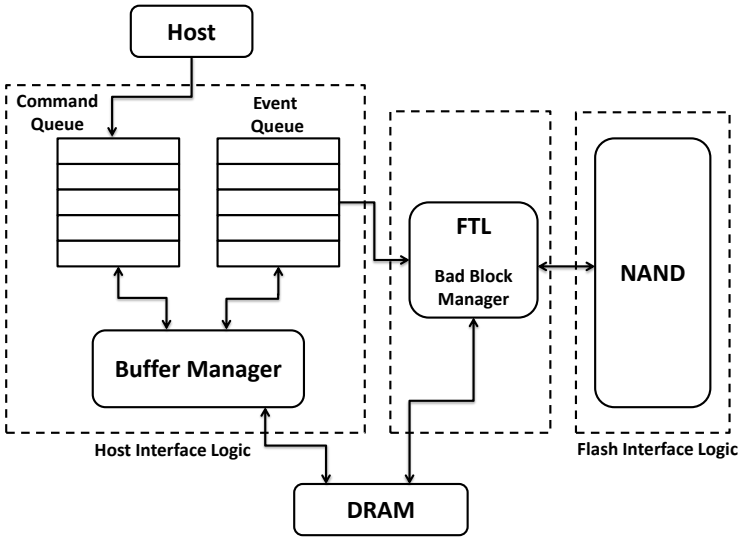


Figure 4.4: **O**penSSD Internals: Major components of OpenSSD internal design are host interface logic, flash interface logic and flash translation layer.

command queues and a buffer manager. The command queues provide a FIFO for incoming requests to decouple FTL operations from receiving SATA requests. The hardware provides separate read and write command queues, into which arriving commands can be placed. The queue provides a *fast path* for performance-sensitive commands. Less common commands, such as *ATA flush*, *idle* and *standby* are executed on a *slow path* that waits for all queued commands to complete. The device transfers data from the host using a separate DMA controller, which copies data between host and device DRAM through a hardware SATA buffer manager (a circular FIFO buffer space).

The device firmware logically consists of three components as shown in Figure 4.4: host interface logic, the FTL, and flash interface logic. The host interface logic decodes incoming commands and either enqueues them in the command queues (for reads and writes), or stalls waiting for queued commands to complete. The FTL implements the logic for processing requests, and invokes the flash interface to actually read, write, copy, or erase flash data. The OpenSSD platform comes with open-source firmware libraries for accessing the hardware and three sample FTLs. We use the page-mapped GreedyFTL (with 32 KB pages) as our baseline as it provides support for garbage collection.

4.4.3 Implementation: Simulator vs. Prototype

We describe the implementation challenges, our experiences, and lessons learned while prototyping SSC on the OpenSSD platform in more detail in Chapter 5. Here, we list the major differences between the simulator and prototype implementations of SSC designs.

- The OpenSSD platform does not provide any access to OOB area or atomic-writes for logging. As a result, we modify the logging protocol of SSC design to use the last page of an erase block to write updates to address map.
- We use a host-triggered checkpointing mechanism because it incurs less interference than periodic checkpointing used in simulation.
- We use a page-map FTL for OpenSSD because it is simpler to implement in firmware than the simulated hybrid FTL. Page-map FTLs are more fine-grained and faster than hybrid FTLs, which are commonly used in commodity SSDs. As a result, we only prototype the SSC variant of silent eviction policy.
- We use a linear hash-map to store the address translation map in device memory on OpenSSD because of no memory allocation support in firmware for implementing the simulated sparse hash-map data structure.

4.5 Evaluation

We compare the cost and benefits of FlashTier's design components against traditional caching on SSDs and focus on three key questions:

- What are the benefits of providing a sparse unified cache address space for FlashTier?
- What is the cost of providing cache consistency and recovery guarantees in FlashTier?
- What are the benefits of silent eviction for free space management and write performance in FlashTier?

Page read/write	65/85 µs	Block erase	1000 µs
Bus control delay	2 µs	Control delay	10 µs
Flash planes	10	Erase block/plane	256
Pages/erase block	64	Page size	4096 bytes
Seq. Read	585 MB/sec	Rand. Read	149,700 IOPS
Seq. Write	124 MB/sec	Rand. Write	15,300 IOPS

Table 4.3: **S**imulation parameters.

- What are the benefits of SSC design for arbitrary workloads on a real hardware prototype? Do they validate simulation results?

We describe our methods and present a summary of our results before answering these questions in detail.

4.5.1 Methodology

Simulation Methodolgy. We simulate an SSC with the parameters in Table 4.3, which are taken from the latencies of the third generation Intel 300 series SSD [47]. We scale the size of each plane to vary the SSD capacity. On the SSD, we over provision by 7% of the capacity for garbage collection. The SSC does not require over provisioning, because it does not promise a fixed-size address space. The performance numbers are not parameters but rather are the measured output of the SSC timing simulator, and reflect performance on an empty SSD/SSC. Other mainstream SSDs documented to perform better rely on deduplication or compression, which are orthogonal to our design [81].

We compare the FlashTier system against the *Native* system, which uses uses the unmodified Facebook FlashCache cache manager and the FlashSim SSD simulator. We experiment with both write-through and write-back modes of caching. The write-back cache manager stores its metadata on the SSD, so it can recover after a crash, while the write-through cache manager cannot.

We use four real-world traces with the characteristics shown in Table 4.4. These traces were collected on systems with different I/O workloads that consist of a departmental email server (*mail* workload), and file server (*homes* workload) [53]; and a small enterprise data

Workload	**R**ange	**U**nique Blocks	**T**otal Ops.	% Writes
homes	532 GB	1,684,407	17,836,701	95.9
mail	277 GB	15,136,141	462,082,021	88.5
usr	530 GB	99,450,142	116,060,427	5.9
proj	816 GB	107,509,907	311,253,714	14.2

Table 4.4: **W**orkload Characteristics: All requests are sector-aligned and 4,096 bytes.

center hosting user home directories (*usr* workload) and project directories (*proj* workload) [74]. Workload duration varies from 1 week (*usr* and *proj*) to 3 weeks (*homes* and *mail*). The range of logical block addresses is large and sparsely accessed, which helps evaluate the memory consumption for address translation. The traces also have different mixes of reads and writes (the first two are write heavy and the latter two are read heavy) to let us analyze the performance impact of the SSC interface and silent eviction mechanisms. To keep replay times short, we use only the first 20 million requests from the *mail* workload, and the first 100 million requests from *usr* and *proj* workloads.

Prototype Methodology. We compare the SSC prototype against a system using only a disk and a system using the optimized OpenSSD as a cache with Facebook's unmodified FlashCache software [28]. We enable *selective caching* in the cache manager, which uses the disk for sequential writes and only sends random writes to the SSC. This feature was already present in Facebook's FlashCache software, upon which we based our cache manager. We evaluate using standard workload profiles with the filebench benchmark.

We use a system with 3 GB DRAM and a Western Digital WD1502FAEX 7200 RPM 1.5 TB plus the OpenSSD board configured with 4 flash modules (128 GB total). We reserve a 75 GB partition on disk for test data, and configure the OpenSSD firmware to use only 8 GB in order to force garbage collection. We compare three systems: *No-Cache* uses only the disk, *SSD* uses unmodified FlashCache software in either write-through (*WT*) or write-back (*WB*) mode running on the optimized baseline SSD (Section 5.1). The *SSC* platform uses our SSC implementation with a version of FlashCache modified to use FlashTier's caching policies, again in write-back or write-through mode.

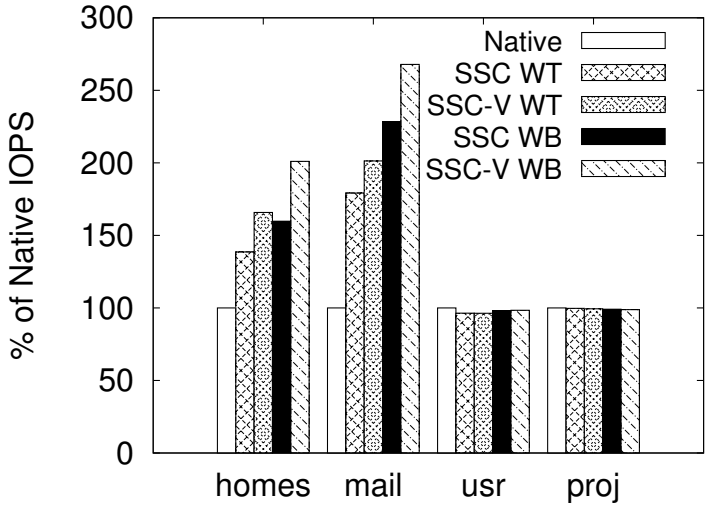


Figure 4.5: **A**pplication Performance: The performance of write-through and write-back FlashTier systems normalized to native write-back performance. We do not include native write-through because it does not implement durability.

4.5.2 Simulated System Comparison

FlashTier improves on caching with an SSD by improving performance, reducing memory consumption, and reducing wear on the device. We begin with a high-level comparison of FlashTier and SSD caching, and in the following sections provide a detailed analysis of FlashTier's behavior.

Performance. Figure 4.5 shows the performance of the two FlashTier configurations with SSC and SSC-V in write-back and write-through modes relative to the native system with an SSD cache in write-back mode. For the write-intensive *homes* and *mail* workloads, the FlashTier system with SSC outperforms the native system by 59-128% in write-back mode and 38-79% in write-through mode. With SSC-V, the FlashTier system outperforms the native system by 101-167% in write-back mode and by 65-102% in write-through mode. The write-back systems improve the performance of cache writes, and hence perform best on these workloads.

For the read-intensive *usr* and *proj* workloads, the native system performs almost identical to the FlashTier system. The performance gain comes largely from garbage collection, as we

describe in Section 4.5.6, which is offset by the cost of FlashTier's consistency guarantees, which are described in Section 4.5.5.

Memory consumption. Table 4.5 compares the memory usage on the device for the native system and FlashTier. Overall, Flashtier with the SSC consumes 11% more device memory and with SSC-V consumes 160% more. However, both FlashTier configurations consume 89% less host memory. We describe these results more in Section 4.5.4.

Wearout. For Figure 4.5, we also compare the number of erases and the wear differential (indicating a skewed write pattern) between the native and FlashTier systems. Overall, on the write-intensive *homes* and *mail* workloads, FlashTier with SSC improves flash lifetime by 40%, and with SSC-V by about 67%. On the read-intensive *usr* and *proj* workloads, there are very few erase operations, and all the three device configurations last long enough to never get replaced. The silent-eviction policy accounts for much of the difference in wear: in write-heavy workloads it reduces the number of erases but in read-heavy workloads may evict useful data that has to be written again. We describe these results more in Section 4.5.6.

4.5.3 Prototype System Comparison

Performance. Figure 4.6 shows the performance of the filebench workloads — fileserver (1:2 reads/writes), webserver (10:1 reads/writes), and varmail (1:1:1 reads/writes/fsyncs) — using SSD and SSC in write-back and write-through caching modes, similar to FlashTier [99]. The performance is normalized to the *No-Cache* system.

First, we find that both SSD and SSC systems significantly outperform disk for all three workloads. This demonstrates that the platform is fast enough to execute real workloads and measure performance improvements. Second, we find that for the write-intensive fileserver workload, the SSC prototype shows benefits of silent eviction. In the write-back configuration, the SSC performs 52% better than the improved baseline SSD. In the write-through configuration, the SSC performs 45% better. For the read-intensive webserver workload, we find that most data accesses are random reads and there is no garbage collection overhead

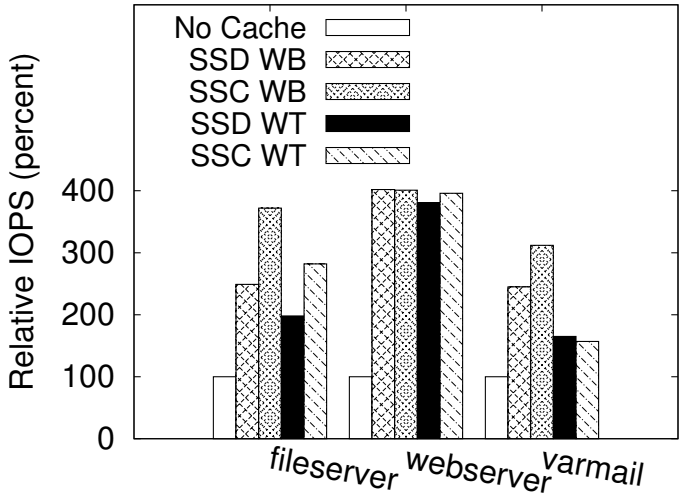


Figure 4.6: **S**SC Prototype Performance. The performance of write-back and write-through caches using SSD and SSC relative to no-cache disk execution.

or silent eviction benefit, which repeats FlashTier projections. In addition, there was little performance loss from moving eviction decisions out of the cache manager and into the FTL.

These tests are substantially different than the ones reported for FlashTier in simulation in Section 4.5.2 (different workloads, page-mapped vs. hybrid FTL, different performance), but the overall results validate that the SSC design does have the potential to greatly improve caching performance. The benefit of the SSC design is lower with OpenSSD largely because the baseline uses a page-mapped FTL instead of a hybrid FTL, so garbage collection is less expensive.

Wearout. Third, we find that silent eviction reduces the number of erase operations, similar to FlashTier's results. For the fileserver workload, the system erases 90% fewer blocks in both write-back and write-through modes. This reduction occurs because silent eviction creates empty space by aggressively evicting clean data that garbage collection keeps around. In addition, silent eviction reduces the average latency of I/O operations by 39% with the SSC write-back mode compared to the SSD and 31% for write-through mode.

Consistency Cost. Overall, our results with OpenSSD correlate with the published FlashTier results. The major difference in functionality between the two systems is different consistency

	Size	**S**SD	**S**SC	**S**SC-V	**N**ative	**F**TCM
Workload	**G**B	**D**evice (MB)			**H**ost (MB)	
homes	1.6	1.13	1.33	3.07	8.83	0.96
mail	14.4	10.3	12.1	27.4	79.3	8.66
usr	94.8	66.8	71.1	174	521	56.9
proj	102	72.1	78.2	189	564	61.5
proj-50	205	144	152	374	1,128	123

Table 4.5: **M**emory Consumption: Total size of cached data, and host and device memory usage for Native and FlashTier systems for different traces. FTCM: write-back FlashTier Cache Manager.

guarantees: FlashTier's SSC synchronously wrote metadata after every *write-dirty*, while our OpenSSD version writes metadata when an erase block fills or after an ATA *flush* command. For the fsync and write-intensive varmail workload, we find that the cost of consistency for the SSC prototype due to logging and checkpoints within the device is lower than the extra synchronous metadata writes from host to SSD. As a result, the SSC prototype performs 27% better than the SSD system in write-back mode. In write-through mode, the SSD system does not provide any data persistence, and outperforms the SSC prototype by 5%.

4.5.4 FlashTier Address Space Management

In this section, we evaluate the device and cache manager memory consumption from using a single sparse address space to maintain mapping information and block state. Our memory consumption results for the SSC prototype are similar to simulation presented in this section. However, we do not prototype the SSC-V device because we only implement the simpler and faster page-map FTL in firmware. As a result, we only show the results from simulation in this section.

Device memory usage. Table 4.5 compares the memory usage on the device for the native system and FlashTier. The native system SSD stores a dense mapping translating from SSD logical block address space to physical flash addresses. The FlashTier system stores a sparse mapping from disk logical block addresses to physical flash addresses using a sparse hash map.

Both systems use a hybrid layer mapping (HLM) mixing translations for entire erase blocks with per-page translations. We evaluate both SSC and SSC-V configurations.

For this test, both SSD and SSC map 93% of the cache using 256 KB blocks and the remaining 7% is mapped using 4 KB pages. SSC-V stores page-level mappings for a total of 20% for reserved space. As described earlier in Section 4.3.3, SSC-V can reduce the garbage collection cost by using the variable log policy to increase the percentage of log blocks. In addition to the target physical address, both SSC configurations store an eight-byte dirty-page bitmap with each block-level map entry in device memory. This map encodes which pages within the erase block are dirty.

We measure the device memory usage as we scale the cache size to accommodate the 25% most popular blocks from each of the workloads, and top 50% for *proj-50*. The SSD averages 2.8 bytes/block, while the SSC averages 3.1 and SSC-V averages 7.4 (due to its extra page-level mappings). The *homes* trace has the lowest density, which leads to the highest overhead (3.36 bytes/bock for SSC and 7.7 for SSC-V), while the *proj-50* trace has the highest density, which leads to lower overhead (2.9 and 7.3 bytes/block).

Across all cache sizes from 1.6 GB to 205 GB, the sparse hash map in SSC consumes only 5–17% more memory than SSD. For a cache size as large as 205 GB for *proj-50*, SSC consumes no more than 152 MB of device memory, which is comparable to the memory requirements of an SSD. The performance advantages of the SSC-V configuration comes at the cost of doubling the required device memory, but is still only 374 MB for a 205 GB cache.

The average latencies for remove and lookup operations are less than 0.8 µs for both SSD and SSC mappings. For inserts, the sparse hash map in SSC is 90% slower than SSD due to the rehashing operations. However, these latencies are much smaller than the bus control and data delays and thus have little impact on the total time to service a request.

Host memory usage. The cache manager requires memory to store information about cached blocks. In write-through mode, the FlashTier cache manager requires no per-block state, so its memory usage is effectively zero, while the native system uses the same amount of memory for both write-back and write-through. Table 4.5 compares the cache-manager

memory usage in write-back mode for native and FlashTier configured with a dirty percentage threshold of 20% of the cache size (above this threshold the cache manager will clean blocks).

Overall, the FlashTier cache manager consumes less than 11% of the native cache manager. The native system requires 22 bytes/block for a disk block number, checksum, LRU indexes and block state. The FlashTier system stores a similar amount of data (without the Flash address) for dirty blocks, but nothing for clean blocks. Thus, the FlashTier system consumes only 2.4 bytes/block, an 89% reduction. For a cache size of 205 GB, the savings with FlashTier cache manager are more than 1 GB of host memory.

Overall, the SSC provides a 78% reduction in total memory usage for the device and host combined. These savings come from the unification of address space and metadata across the cache manager and SSC. Even with the additional memory used for the SSC-V device, it reduces total memory use by 60%. For systems that rely on host memory to store mappings, such as FusionIO devices [1], these savings are immediately realizable.

4.5.5 FlashTier Consistency

In this section, we evaluate the cost of crash consistency and recovery by measuring the simulated overhead of logging, checkpointing and the time to recover. On a system with non-volatile memory or that can flush RAM contents to flash on a power failure, consistency imposes no performance cost because there is no need to write logs or checkpoints.

We have shown the cost of crash-consistency for prototyping in Section 4.5.3. In the prototype, we have no access to OOB area and atomic-writes. As a result, our logging and checkpointing protocols are different from the simulated design. Our prototype implementation uses application-defined checkpointing points for its consistency semantics.

Consistency cost. We first measure the performance cost of FlashTier's consistency guarantees by comparing against a baseline *no-consistency* system that does not make the mapping persistent. Figure 4.7 compares the throughput of FlashTier with the SSC configuration and the native system, which implements consistency guarantees by writing back mapping metadata, normalized to the no-consistency system. For FlashTier, we configure group commit

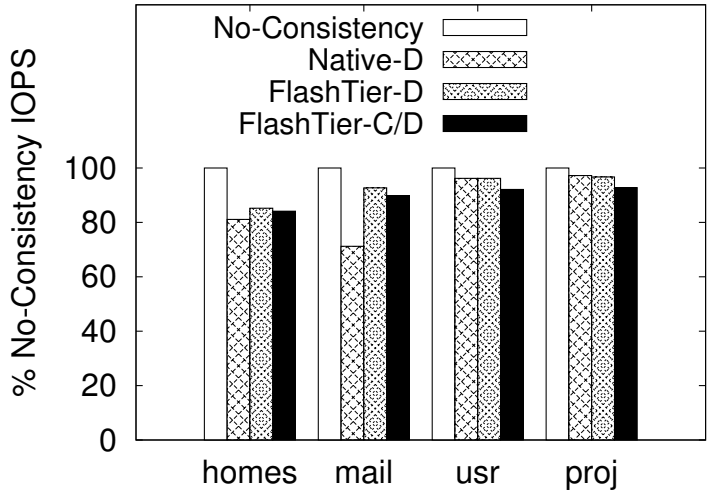


Figure 4.7: **C**onsistency Cost: No-consistency system does not provide any consistency guarantees for cached data or metadata. Native-D and FlashTier-D systems only provide consistency for dirty data. FlashTier-C/D provides consistency for both clean and dirty data.

to flush the log buffer every 10,000 write operations or when a synchronous operation occurs. In addition, the SSC writes a checkpoint if the log size exceeds two-thirds of the checkpoint size or after 1 million writes, whichever occurs earlier. This limits both the number of log records flushed on a commit and the log size replayed on recovery. For the native system, we assume that consistency for mapping information is provided by out-of-band (OOB) writes to per-page metadata without any additional cost [3].

As the native system does not provide persistence in write-through mode, we only evaluate write-back caching. For efficiency, the native system (Native-D) only saves metadata for dirty blocks at runtime, and loses clean blocks across unclean shutdowns or crash failures. It only saves metadata for clean blocks at shutdown. For comparison with such a system, we show two FlashTier configurations: FlashTier-D, which relaxes consistency for clean blocks by buffering log records for *write-clean*, and FlashTier-C/D, which persists both clean and dirty blocks using synchronous logging.

For the write-intensive *homes* and *mail* workloads, the extra metadata writes by the native cache manager to persist block state reduce performance by 18-29% compared to the no-consistency system. The *mail* workload has 1.5x more metadata writes per second

than *homes*, therefore, incurs more overhead for consistency. The overhead of consistency for persisting clean and dirty blocks in both FlashTier systems is lower than the native system, at 8-15% for FlashTier-D and 11-16% for FlashTier-C/D. This overhead stems mainly from synchronous logging for insert/remove operations from *write-dirty* (inserts) and *write-clean* (removes from overwrite). The *homes* workload has two-thirds fewer *write-clean* operations than *mail*, and hence there is a small performance difference between the two FlashTier configurations.

For read-intensive *usr* and *proj* workloads, the cost of consistency is low for the native system at 2-5%. The native system does not incur any synchronous metadata updates when adding clean pages from a miss and batches sequential metadata updates. The FlashTier-D system performs identical to the native system because the majority of log records can be buffered for *write-clean*. The FlashTier-C/D system's overhead is only slightly higher at 7%, because clean writes following a miss also require synchronous logging.

We also analyze the average request response time for both the systems. For write-intensive workloads *homes* and *mail*, the native system increases response time by 24-37% because of frequent small metadata writes. Both FlashTier configurations increase response time less, by 18-32%, due to logging updates to the map. For read-intensive workloads, the average response time is dominated by the read latencies of the flash medium. The native and FlashTier systems incur a 3-5% increase in average response times for these workloads respectively. Overall, the extra cost of consistency for the request response time is less than 26 µs for all workloads with FlashTier.

Recovery time. Figure 4.8 compares the time for recovering after a crash. The mapping and cache sizes for each workload are shown in Table 4.5.

For FlashTier, the only recovery is to reload the mapping and block state into device memory. The cache manager metadata can be read later. FlashTier recovers the mapping by replaying the log on the most recent checkpoint. It recovers the cache manager state in write-back mode using *exists* operations. This is only needed for space management, and thus can be deferred without incurring any start up latency. In contrast, for the native system

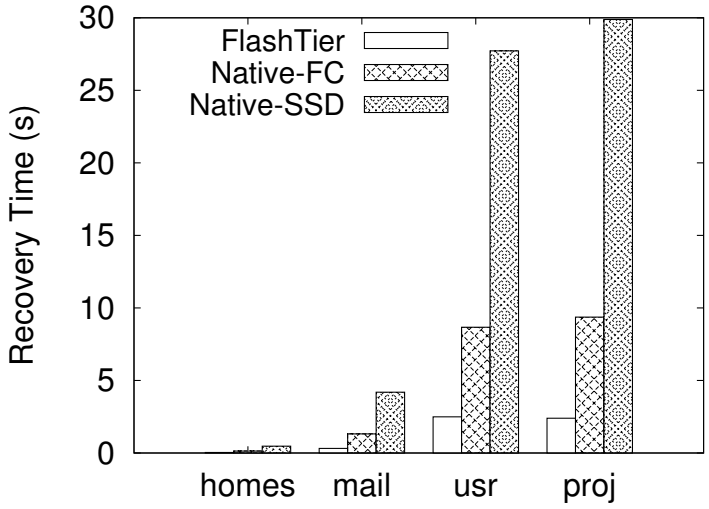


Figure 4.8: **R**ecovery Time: Native-FC accounts for only recovering FlashCache cache manager state. Native-SSD accounts for only recovering the SSD mapping.

both the cache manager and SSD must reload mapping metadata.

Most SSDs store the logical-to-physical map in the OOB area of a physical page. We assume that writing to the OOB is free, as it can be overlapped with regular writes and hence has little impact on write performance. After a power failure, however, these entries must be read to reconstruct the mapping [3], which requires scanning the whole SSD in the worst case. We estimate the best case performance for recovering using an OOB scan by reading just enough OOB area to equal the size of the mapping table.

The recovery times for FlashTier vary from 34 ms for a small cache (*homes*) to 2.4 seconds for *proj* with a 102 GB cache. In contrast, recovering the cache manager state alone for the native system is much slower than FlashTier and takes from 133 ms for *homes* to 9.4 seconds for *proj*. Recovering the mapping in the native system is slowest because scanning the OOB areas require reading many separate locations on the SSD. It takes from 468 ms for *homes* to 30 seconds for *proj*.

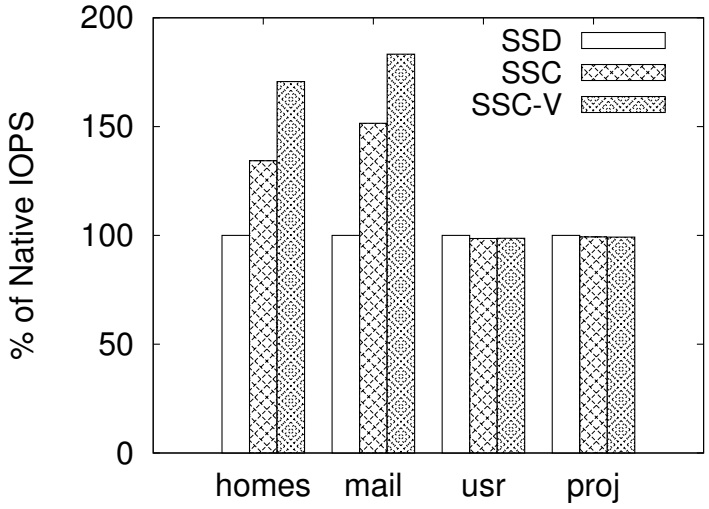


Figure 4.9: **G**arbage Collection Performance: Comparing the impact of garbage collection on caching performance for different workloads on SSD, SSC and SSC-V devices.

4.5.6 FlashTier Silent Eviction

In this section, we evaluate the impact of silent eviction on caching performance and wear management in simulation. We compare the behavior of caching on three devices: SSD, SSC and SSC-V, which use garbage collection and silent eviction with fixed and variable log space policies. For all traces, we replay the trace on a cache sized according to Table 4.5. To warm the cache, we replay the first 15% of the trace before gathering statistics, which also ensures there are no available erased blocks. To isolate the performance effects of silent eviction, we disabled logging and checkpointing for these tests and use only write-through caching, in which the SSC is entirely responsible for replacement.

Our full system results for silent eviction on the OpenSSD prototype with the SSC device policy have been shown in Section 4.5.3. Those end-to-end results validate our findings for simulation in this section.

Garbage Collection. Figure 4.9 shows the performance impact of silent eviction policies on SSC and SSC-V. We focus on the write-intensive *homes* and *mail* workloads, as the other two workloads have few evictions. On these workloads, the Flashtier system with SSC

	Write Amp.			Miss Rate		
Workload	SSD	SSC	SSC-V	SSD	SSC	SSC-V
homes	2.30	1.84	1.30	10.4	12.8	11.9
mail	1.96	1.08	0.77	15.6	16.9	16.5
usr	1.23	1.30	1.18	10.6	10.9	10.8
proj	1.03	1.04	1.02	9.77	9.82	9.80

Table 4.6: Cache Performance: For each workload, the write amplification, and the cache miss rate is shown for SSD, SSC and SSC-V.

	Erases			Wear Difference		
Workload	SSD	SSC	SSC-V	SSD	SSC	SSC-V
homes	878,395	829,356	617,298	3,094	864	431
mail	880,710	637,089	525,954	1,044	757	181
usr	339,198	369,842	325,272	219	237	122
proj	164,807	166,712	164,527	41	226	17

Table 4.7: Wear Distribution: For each workload, the total number of erase operations and the maximum wear difference between blocks is shown for SSD, SSC and SSC-V.

outperforms the native SSD by 34-52%, and SSC-V by 71-83%. This improvement comes from the reduction in time spent for garbage collection because silent eviction avoids reading and rewriting data. This is evidenced by the difference in write amplification, shown in Table 4.6. For example, on *homes*, the native system writes each block an additional 2.3 times due to garbage collection. In contrast, with SSC the block is written an additional 1.84 times, and with SSC-V, only 1.3 more times. The difference between the two policies comes from the additional availability of log blocks in SSC-V. As described in Section 4.3.3, having more log blocks improves performance for write-intensive workloads by delaying garbage collection and eviction, and decreasing the number of valid blocks that are discarded by eviction. The performance on *mail* is better than *homes* because the trace has 3 times more overwrites per disk block, and hence more nearly empty erase blocks to evict.

Cache Misses. The time spent satisfying reads is similar in all three configurations across all four traces. As *usr* and *proj* are predominantly reads, the total execution times for these traces is also similar across devices. For these traces, the miss rate, as shown in Table 4.6, increases negligibly.

On the write-intensive workloads, the FlashTier device has to impose its policy on what to replace when making space for new writes. Hence, there is a larger increase in miss rate, but in the worst case, for *homes*, is less than 2.5 percentage points. This increase occurs because the SSC eviction policy relies on erase-block utilization rather than recency, and thus evicts blocks that were later referenced and caused a miss. For SSC-V, though, the extra log blocks again help performance by reducing the number of valid pages evicted, and the miss rate increases by only 1.5 percentage points on this trace. As described above, this improved performance comes at the cost of more device memory for page-level mappings. Overall, both silent eviction policies keep useful data in the cache and greatly increase the performance for recycling blocks.

Wear Management. In addition to improving performance, silent eviction can also improve reliability by decreasing the number of blocks erased for merge operations. Table 4.7 shows the total number of erase operations and the maximum wear difference (indicating that some blocks may wear out before others) between any two blocks over the execution of different workloads on SSD, SSC and SSC-V.

For the write-intensive *homes* and *mail* workloads, the total number of erases reduce for SSC and SSC-V. In addition, they are also more uniformly distributed for both SSC and SSC-V. We find that most erases on SSD are during garbage collection of *data blocks* for copying valid pages to free log blocks, and during full merge operations for recycling *log blocks*. While SSC only reduces the number of copy operations by evicting the data instead, SSC-V provides more log blocks. This reduces the total number of full merge operations by replacing them with switch merges, in which a full log block is made into a data block. On these traces, SSC and SSC-V improve the flash lifetime by an average of 40% and 67%, and the overhead of copying valid pages by an average of 32% and 52% respectively, as compared to the SSD. The SSC device lasts about 3.5 years and SSC-V lasts about 5 years, after which they can be replaced by the system administrator.

For the read-intensive *usr* and *proj* workloads, most blocks are read-only, so the total number of erases and wear difference is lower for all three devices. As a result, all three devices

last more than 150 years, so as to never get replaced. However, the SSC reduces flash lifetime by about 21 months because it evicts data that must later be brought back in and rewritten. However the low write rate for these traces makes reliability less of a concern. For SSC-V, the lifetime improves by 4 months, again from reducing the number of merge operations.

Both SSC and SSC-V greatly improve performance and on important write-intensive workloads, also decrease the write amplification and the resulting erases. Overall, the SSC-V configuration performs better, has a lower miss rate, and better reliability and wear-leveling achieved through increased memory consumption and a better replacement policy.

4.6 Related Work

The FlashTier design draws on past work investigating the use of solid-state memory for caching and hybrid systems.

SSD Caches. Guided by the price, power and performance of flash, cache management on flash SSDs has been proposed for fast access to disk storage. Windows and Solaris have software cache managers that use USB flash drives and solid-state disks as read-optimized disk caches managed by the file system [8, 35]. Oracle has a write-through flash cache for databases [84] and Facebook has started the deployment of their in-house write-back cache manager to expand the OS cache for managing large amounts of data on their Timeline SQL servers [28, 96]. Storage vendors have also proposed the use of local SSDs as write-through caches to centrally-managed storage shared across virtual machine servers [14, 76]. However, all these software-only systems are still limited by the narrow storage interface, multiple levels of address space, and free space management within SSDs designed for persistent storage. In contrast, FlashTier provides a novel consistent interface, unified address space, and silent eviction mechanism within the SSC to match the requirements of a cache, yet maintaining complete portability for applications by operating at block layer.

Hybrid Systems. SSD vendors have recently proposed new flash caching products, which cache most-frequently accessed reads and write I/O requests to disk [29, 80]. FlashCache [51] and the flash-based disk cache [94] also propose specialized hardware for caching. Hybrid

drives [10] provision small amounts of flash caches within a hard disk for improved performance. Similar to these systems, FlashTier allows custom control of the device over free space and wear management designed for the purpose of caching. In addition, FlashTier also provides a consistent interface to persist both clean and dirty data. Such an interface also cleanly separates the responsibilities of the cache manager, the SSC and disk, unlike hybrid drives, which incorporate all three in a single device. The FlashTier approach provides more flexibility to the OS and applications for informed caching.

Informed Caching. Past proposals for multi-level caches have argued for informed and exclusive cache interfaces to provide a single, large unified cache in the context of storage arrays [115, 113]. Recent work on storage tiering and differentiated storage services has further proposed to classify I/O and use different policies for cache allocation and eviction on SSD caches based on the information available to the OS and applications [69, 36]. However, all these systems are still limited by the narrow storage interface of SSDs, which restricts the semantic information about blocks available to the cache. The SSC interface bridges this gap by exposing primitives to the OS for guiding cache allocation on writing clean and dirty data, and an explicit *evict* operation for invalidating cached data.

Storage Interfaces. Recent work on a new nameless-write SSD interface and virtualized flash storage for file systems have argued for removing the costs of indirection within SSDs by exposing physical flash addresses to the OS [116], providing caching support [75], and completely delegating block allocation to the SSD [50]. Similar to these systems, FlashTier unifies multiple levels of address space, and provides more control over block management to the SSC. In contrast, FlashTier is the first system to provide internal flash management and a novel device interface to match the requirements of caching. Furthermore, the SSC provides a virtualized address space using disk logical block addresses, and keeps its interface grounded within the SATA read/write/trim space without requiring migration callbacks from the device into the OS like these systems.

4.7 Summary

Flash caching promises an inexpensive boost to storage performance. However, traditional SSDs are designed to be a drop-in disk replacement and do not leverage the unique behavior of caching workloads, such as a large, sparse address space and clean data that can safely be lost. In this paper, we describe FlashTier, a system architecture that provides a new flash device, the SSC, which has an interface designed for caching. FlashTier provides memory-efficient address space management, improved performance and cache consistency to quickly recover cached data following a crash. As new non-volatile memory technologies become available, such as phase-change and storage-class memory, it will be important to revisit the interface and abstraction that best match the requirements of their memory tiers.

5 SSC PROTOTYPE: EXPERIENCES & LESSONS

Due to the high performance, commercial success, and complex internal mechanisms of solid-state drives (SSDs), there has been a great deal of work on optimizing their use (*e.g.*, caching [69, 82]), optimizing their internal algorithms (*e.g.*, garbage collection [17, 3, 38]), and extending their interface (*e.g.*, caching operations [99]). However, most research looking at internal SSD mechanisms relies on simulation rather than direct experimentation [5, 82, 99, 116].

Thus, there is little known about real-world implementation trade-offs relevant to SSD design, such as the cost of changing their command interface. Most such knowledge has remained the intellectual property of SSD manufacturers [48, 80, 29, 30], who release little about the internal workings of their devices. This limits the opportunities for research innovation on new flash interfaces, on OS designs to better integrate flash in the storage hierarchy, and on adapting the SSD internal block management for application requirements.

Simulators and emulators suffer from two major sources of inaccuracy. First, they are limited by the quality of performance models, which may miss important real-world effects. Second, simulators often simplify systems and may leave out important components, such as the software stack used to access an SSD.

We sought to validate the FlashTier's Solid-State Cache (SSC) by implementing it as a prototype on the OpenSSD Jasmine hardware platform [107]. In this chapter, we describe the challenges faced by us while prototyping the SSC design, our experiences, and general lessons learned, which are applicable to new SSD designs and flash interfaces.

First, we found that the baseline performance of the OpenSSD board was low, and that the effect of new designs was drowned out by other performance problems. In Section 5.1, we describe our experiences with the introduction of a *merge buffer* to align writes to internal boundaries and a *read buffer* for efficient random reads.

Second, we found that passing new commands from the file-system layer through the Linux storage stack and into the device firmware raised substantial engineering hurdles. For example, the I/O scheduler must know which commands can be merged and reordered. In

Section 5.2, we describe the novel techniques we developed to tunnel new commands through the storage stack and hardware as variations of existing commands, which limits software changes to the layers at the ends of the tunnel. For example, we return cache-miss errors in the data field of a read.

Third, we encountered practical hardware limitations within the firmware. The OpenSSD board lacks an accessible out-of-band (OOB) area, and has limited SRAM and DRAM within the device. Section 5.3 describes how we redesigned the mechanisms and semantics for providing consistency and durability guarantees in an SSC. We change the semantics to provide consistency only when demanded by higher level-software via explicit commands.

Our goal was to validate the design claims made by FlashTier as well as to gain experience prototyping SSD designs. We develop our prototypes for Linux kernel version 2.6.33, and much of the effort focused on integrating the new designs into Linux's storage stack.

We next describe the challenges and problems faced during the prototyping process, our solutions and general lessons applicable to new SSD and interface designs.

5.1 Baseline Performance

The biggest challenge with prototyping the SSC design was to first improve the performance of the baseline platform to its maximum. This enables us to run real workloads and get realistic results comparable to commercial SSDs. The OpenSSD hardware is capable of 90 MB/sec writes and 125 MB/sec reads. However, the reference FTL implementations delivered this performance only with specific workloads, as we describe below. For common cases such as 4 KB random writes, performance dropped to 10 MB/sec. Thus, any performance benefit we achieved through SSC would be overwhelmed by poor baseline performance. The first task of prototyping a new interface to flash, ironically, was to make the existing interface perform well. We note that many of the challenges we faced are known, and here we describe the effort required to implement solutions.

Workload **R**equest Size	**B**aseline **I**O/s	**O**S Req. **S**ize (sectors)	**R**oot **C**ause	**I**O/s **I**ntel 520	 **D**isk
4 KB	2,560	9	Partial	22,528	1,497
8 KB	2,048	18	Partial	17,280	826
16 KB	1,920	36	Partial	11,008	563
32 KB	2,912	82	Align	7,136	419
64 KB	1,456	150	Par.	3,824	288
128 KB	648	256	Par.	2,056	212
256 KB	248	257	Cont.	988	149
512 KB	132	266	Cont.	500	98
Seq 4 KB	15,616	336	Partial	61,950	24,437
Seq 4+32 KB	20,736	64	Align	62,120	22,560

Table 5.1: **W**rite Performance: Random write performance (top) and sequential write performance (bottom) as compared to an Intel 520 SSD and a disk. Columns 3 shows the average number of 512 B sectors per request sent to the device, and Column 4 summarizes the cause of poor performance (*Partial* = partial writes, *Align* = unaligned writes, *Par* = insufficient parallelism, *Cont* = bank contention).

5.1.1 Problems

We began by analyzing the peak performance achievable by the hardware and comparing the application performance delivered by the FTL for different workload parameters. We perform tests on an 8 GB partition on an empty device. For read tests, data was written sequentially. Table 5.1 shows I/O Operations/sec (IOPS) with varying random and sequential request sizes in column 2. All measurements were made with the *fio* microbenckmarking tool [66]. For comparison, the last two columns show performance of the Intel 520 120 GB SSD, and for a Western Digital WD1502FAEX 7200 RPM 1.5 TB disk. Overall, random IOPS peaks at 32 KB, the virtual page size of the device, and degrades with larger requests. Smaller requests also suffer reduced IOPS and much lower bandwidth. Read performance, shown in Figure 5.2 was 125 MB/sec for sequential reads but only 2,340 IOPS for random reads.

Alignment and Granularity. The smallest unit of I/O within the FTL is a 32 KB virtual page. For smaller requests, the firmware has to read the old data, merge in the new data, and finally write the updated virtual page. Therefore, writes smaller than 32 KB suffer write

amplification from this read-modify-write cycle. For example, 8 KB blocks achieve only 2,048 IOPS (16 MB/s), while 32 KB blocks achieve 2,912 (91 MB/s). Furthermore, writes that are not aligned on 32 KB boundaries can suffer a read-modify-write cycle for *two* blocks.

Parallelism. While requests aligned and equal to the virtual page size perform well, larger requests degrade performance. With 512 KB requests, performance degrades from 91 MB/sec to 67.5 MB/sec. Here, we identified the problem as internal contention: the controller breaks the request into 32 KB chunks, each of which is sent concurrently to different banks. Because there is no coordination between the writes, multiple writes may occur to the same bank at once, leading to contention.

OS Merging. A final source of problems is the I/O scheduler in Linux. As noted above, large sequential requests degrade performance. Linux I/O schedulers — NOOP, CFQ and Deadline — merge spatially adjacent requests. As shown in column 3 (Req Size) of Table 5.1, for the Seq 4 KB row the OS merges an average of 336 sectors (168 KB) for sequential 4 KB writes and achieves only 61 MB/sec. When we artificially limit merging to 32 KB (64 sectors), shown in the last row (labeled Seq 4 KB+32), performance improves to 81 MB/sec. This is still below peak rate because the requests may not all be aligned to virtual page boundaries.

Overall, this analysis led us to develop techniques that reduce the number of partial writes and to avoid contention from large requests for maximum parallelism. While the OS I/O scheduler can merge spatially adjacent requests for sequential workloads, we develop techniques to achieve this goal for random patterns as well.

5.1.2 Solutions

Write performance. We address poor write performance by implementing a *merge buffer* within the FTL to stage data before writing it out. This ensures that data can be written at virtual-page granularity rather than at the granularity supplied by the operating system. Data for write commands are enqueued in a FIFO order in the SATA write buffer, from which the FTL copies it to a merge buffer. The FTL flushes the buffer only when the buffer is full to ensure there are no partial writes. The buffer is striped across different flash banks to

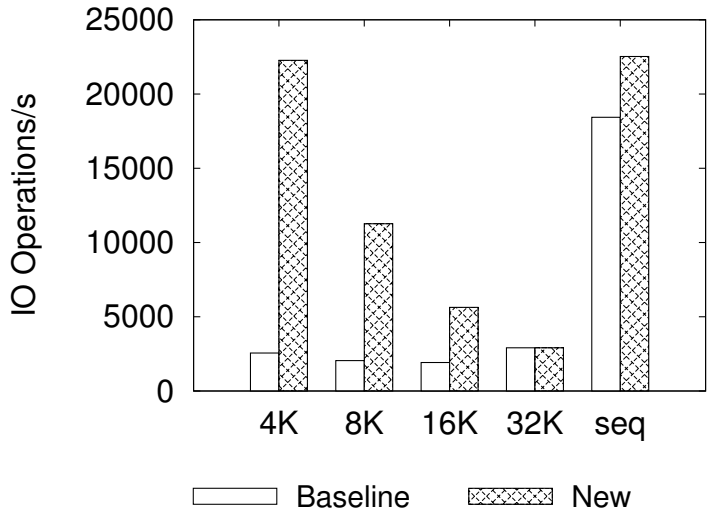


Figure 5.1: **S**mall Requests: Impact of merge buffering on write performance.

minimize the idle time when a request waits for a free bank. This also improves sequential write performance by nullifying the impact of partial or unaligned writes for requests merged by the I/O scheduler.

This change converts the FTL to do log-structured writes. As a result, page address mapping can no longer be done at 32 KB granularity. We modified the FTL to keep mappings at 4 KB, which quadruples the size of the mapping data structure.

Figure 5.1 shows the write performance of the OpenSSD board using the reference firmware and page-mapped FTL implementation (baseline), and with merging and buffering optimizations (new) on an empty device. For the enhanced system, the random write performance for 4 KB requests from the application is *nine times* the baseline, and only 1% lower than sequential writes. Sequential performance is 22% better than the baseline because all writes are performed as aligned 32 KB operations. With the use of a write merge buffer, random write performance is close to the peak device write rate of 22,528 IOPS. This is close to commercial SSD performance and makes the OpenSSD board useful for prototyping.

Read Performance. Sequential read performance of the baseline FTL is excellent (125 MB/sec) because virtual pages contained contiguous data. However, random reads perform similar to random writes and achieve only 2,340 IOPS. Using a merge buffer can hurt sequential reads if

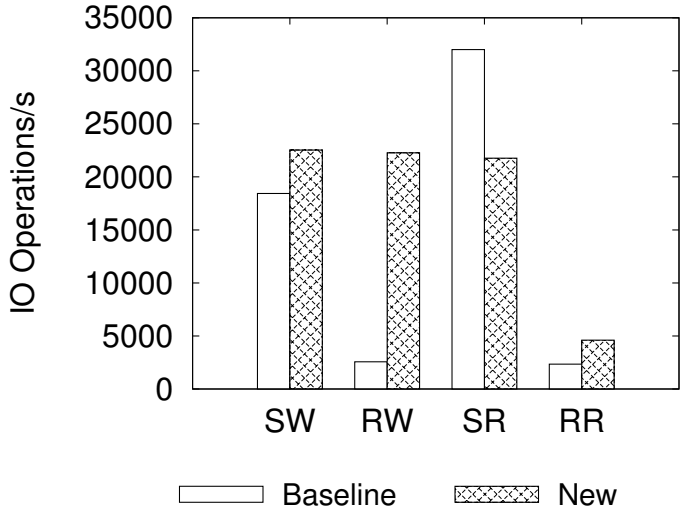


Figure 5.2: **R**ead/Write Performance: On baseline unoptimized and new optimized OpenSSD FTLs for fio benchmark with 4 KB request sizes.

data is written randomly, because a 32 KB virtual page may contain a random selection of data.

We implement two optimizations to improve performance. First, we introduce a *read buffer* between flash and the SATA buffer manager. When a virtual page contains many 4 KB blocks that are valid, the FTL reads the entire virtual page into the read buffer but discards the invalid chunks. For pages that have only a small portion overwritten, this is much more efficient than reading every 4 KB chunk separately. Second, we modified the FTL to start the DMA transfer to the host as soon as the required chunks were in the read buffer, rather than waiting for the whole virtual page to come directly from flash. This reduces the waiting time for partial flash reads. These two optimizations help us to exploit parallelism across different flash banks, and still make sure that the buffer manager starts the DRAM to host transfer only after firmware finishes the flash to DRAM transfer.

Figure 5.2 shows the overall performance of the baseline and new systems for 4 KB requests. For writes, the new system has improved performance for both random and sequential access patterns as shown in Figure 5.1. For sequential reads, the new system is 30% slower because of the delay added by the read buffer. For random 4 KB requests, the new system is twice as

fast as the baseline, largely because of the optimization to reply as soon as only the requested data has been copied. The baseline system, in contrast, waits for an entire 32 KB virtual page on every 4 KB read. Thus, we sacrifice some sequential performance to improve random performance, which is likely to be more beneficial for a cache.

5.1.3 Lessons Learned

The experience analyzing and improving baseline SSD performance reiterated past lessons about SSD design:

- Huge performance gains are available through merging and buffering; designs that forgo these opportunities may suffer on real hardware. With larger block sizes and increased parallelism in upcoming devices, these techniques would be even more important to fully saturate the internal bandwidth provided by flash.
- SSD designs that are sensitive to alignment and request length may have poor performance due to I/O scheduler merging or application workload characteristics. A good SSD design will compensate for poor characteristics of the request stream.

5.2 OS and Device Interfaces

A key challenge in implementing the SSC is that its design change the *interface* to the device by adding new commands and new reverse command responses. In simulation, these calls were easy to add as private interfaces into the simulator. Within Linux though, we had to integrate these commands into the existing storage architecture.

5.2.1 Problems

We identified three major problems while implementing support for SSC in the OS and device: how to get new commands through the OS storage stack into the device, how to get new responses back from the device, and how to implement commands within the device given its hardware limitations.

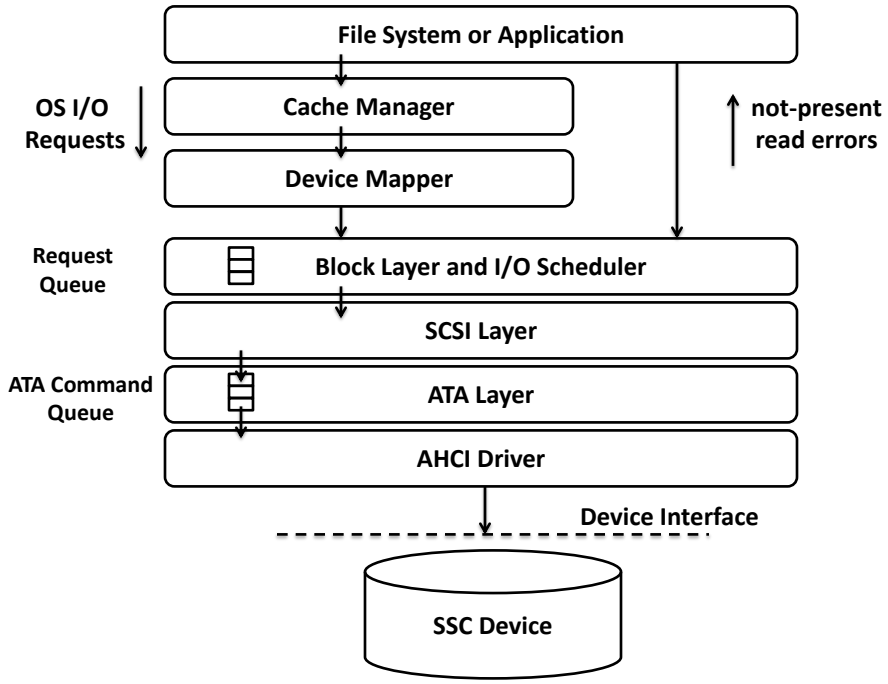


Figure 5.3: **O**S Interfaces: I/O path from the cache manager or file system to the device through different storage stack layers.

First, the forward commands from the OS to the device pass through several layers in the OS, shown in Figure 5.3, which interpret and act on each command differently. Requests enter the storage stack from the file system or layers below, where they go through a scheduler and then SCSI and ATA layers before the AHCI driver submits them to the device. For example, the I/O scheduler can merge requests to adjacent blocks. If it is not aware that the SSC's *write-clean* and *write-dirty* commands are different, it may incorrectly merge them into a single, larger request. Thus, the I/O scheduler layer must be aware of each distinct command.

Second, the reverse path responses from the device to the OS are difficult to change. For example, the SSC interface returns a *not-present* error in response to reading data not in the cache. However, the AHCI driver and ATA layer both interpret error responses as a sign of data loss or corruption. Their error handlers retry the read operation again with the goal of retrieving the page, and then freeze the device by resetting the ATA link. Past research demonstrated that storage systems often retry failed requests automatically [37, 90].

Third, the OpenSSD platform provides hardware support for the SATA protocol (see Figure 4.4) in the form of hardware command queues and a SATA buffer manager. When using the command queues, the hardware does not store the command itself and identifies the command type from the queue it is in. While firmware can choose where and what to enqueue, it can only enqueue two fields: the logical block address (*lba*) and request length (*numsegments*). Furthermore, there are only two queues (read and write), so only two commands can execute as fast commands.

5.2.2 Solutions

We developed several general techniques for introducing new commands. We defer discussion of the detailed mechanisms for SSC to the following section.

Forward commands through the OS. At the block-interface layer, we sought to leave as much code as possible unmodified. Thus, we augment block requests with an additional command field, effectively adding our new commands as sub-types of existing commands. We modified the I/O scheduler to only merge requests with the same command and sub-type. For the SSC, a *write-dirty* command is not merged with a *write-clean* operation. The SCSI or ATA layers blindly pass the sub-type field down to the next layer. We pass all commands that provide data as a write, and all other commands, such as *exists* and *evict* for SSCs, as read commands.

We also modified the AHCI driver to communicate commands to the OpenSSD device. Similar to higher levels, we use the same approach of adding a sub-type to existing commands. Requests use normal SATA commands and pass the new request type in the *rsv1* reserved field, which is set to zero by default.

OpenSSD request handling. Within the device, commands arrive from the SATA bus and are then enqueued by the host-interface firmware. The FTL asynchronously pulls requests from the queues to be processed. Thus, the key change needed for new requests is to communicate the command type from arriving commands to the FTL, which executes commands. We borrow two bits from the length field of the request (a 32-bit value) to encode the command

type. The FTL decodes these length bits to determine which command to execute, and invokes the function for the command. This encoding ensures that the OpenSSD hardware uses the fast path for new variations of read and writes, and allows multiple variations of the commands.

Reverse-path device responses. The key challenge for the SSC implementation is to indicate that a read request missed in the cache without returning an error, which causes the AHCI and ATA layers to retry the request or shutdown the device. Thus, we chose to return a read miss in the data portion of a read as a distinguished pattern; the FTL copies the pattern into a buffer that is returned to the host rather than reading data from flash. The cache manager system software accessing the cache can then check whether the requested block matches the read-miss pattern, and if so consult the backing disk. This approach is not ideal, as it could cause an unnecessary cache miss if a block using the read-miss pattern is ever stored. In addition, it raises the cost of misses, as useless data must be transferred from the device.

5.2.3 Lessons Learned

Passing new commands to the OpenSSD and receiving new responses proved surprisingly difficult.

- The OS storage stack's layered design may require each layer to act differently for the introduction of a new forward command. For example, new commands must have well-defined semantics for request schedulers, such as which commands can be combined and how they can be reordered.

- SSDs hardware and firmware must evolve to support new interfaces. For example, the hardware command queue and the DMA controller are built for standard interfaces and protocols, which requires us to mask the command type in the firmware itself to still use hardware accelerated servicing of commands.

- The device response paths in the OS are difficult to change, so designs that radically

extend existing communication from the device should consider data that will be communicated. For example, most storage code assumes that errors are rare and catastrophic. Interfaces with more frequent errors, such as a cache that can miss, must address how to return benign failures. It may be worthwhile to investigate overloading such device responses on data path for SATA/SAS devices.

5.3 SSC Implementation

The SSC implementation consists of two major portions: the FTL functions implementing the interface, such as *write-dirty* and *write-clean*, and the internal FTL mechanisms implementing FTL features, such as consistency and a unified address space. The interface functions proved simple to implement and often a small extension to the existing FTL functions. In contrast, implementing the internal features was far more difficult.

5.3.1 Interfaces

The data access routines, *read*, *write-dirty* and *write-clean* are similar to existing FTL routines. As noted above, the major difference is that the FTL tracks the clean/dirty status of each page of data, and the *read* command tests to see if the page is present and can fail if it is not. Hence, a *read* request checks the mapping table in the FTL (described in Section 5.3.2) to see if the page is present before fetching the physical location of the data and initiating a flash read. The *write-dirty* and *write-clean* select a physical location to write the data, then update the page map with the new address and a flag indicating if the bit is dirty. Finally, they initiate a flash transfer to write out the new data. We discuss how this data is kept consistent below.

The cache-management commands *evict* and *clean* operate largely on the map: *evict* removes a mapping and marks the corresponding flash page as empty, while *clean* only clears the dirty bit in the map. The *exists* command is used to query the state of a range of logical block addresses, and returns which blocks are dirty and need to be cleaned. The SSC accesses

the page map to check the dirty bit of each page in the range, and return the result as if it was data from a read operation.

5.3.2 Internal Mechanisms

The SSC design describes three features that differ from standard flash FTLs. First, SSCs provide *strong consistency guarantees* on metadata for cache eviction. Second, SSCs present a *unified address space* in which the device can be accessed with disk block addresses. Finally, SSCs implement *silent eviction* as an alternative to garbage collection. We now describe the problems and solutions associated with implementing these mechanisms.

Consistency and durability. The FlashTier SSC design provided durability and consistency for metadata by logging mapping changes to the out-of-band (OOB) area on flash pages. This design was supposed to reduce the latency of synchronous operations, because metadata updates execute with data updates at no additional cost. We found, though, that the OpenSSD hardware reserves the OOB area for error-correcting code and provides no access to software. In addition, the SSC design assumed that checkpoint traffic could be interleaved with foreground traffic, while we found they interfere.

We changed the logging mechanism to instead use the last virtual page of each 4 MB erase block. The FTL maintains a log of changes to the page map in SRAM. After each erase block fills, the FTL writes out the metadata to its last page. This approach does not provide the immediate durability of OOB writes, but amortizes the cost of logging across an entire erase block.

FlashTier uses checkpoints to reduce the time to reconstruct a page map on startup. We store checkpoints in a dedicated area on each flash bank. During a checkpoint, the FTL writes all metadata residing in SSC SRAM and DRAM to the first few erase blocks of each flash bank. While storing metadata in a fixed location could raise reliability issues, they could be reduced by moving the checkpoints around and storing a pointer to the latest checkpoint. The FTL restores the page map from the checkpoint and log after a restart. It loads the

checkpointed SSC SRAM and DRAM segments and then replays the page map updates logged after the checkpoint.

While the FlashTier design wrote out checkpoints on fixed schedule, our implementation defers checkpointing until requested by the host. When an application issues an *fsync* operation, the file system passes this to the device as an ATA *flush* command. We trigger a checkpoint as part of flush. Unfortunately, this can make *fsync()* slower. The FTL also triggers a checkpoint when it receives an ATA *standby* or *idle* command, which allows checkpointing to occur when there are no I/O requests.

Address mappings. The FlashTier cache manager addresses the SSC using disk logical block numbers, which the SSC translates to physical flash addresses. However, this unification introduces a high degree of sparseness in the SSC address space, since only a small fraction of the disk blocks are cached within the SSC. The FlashTier design supported sparse address spaces with a memory-optimized data structure based on perfect hash function [34]. This data structure used dynamically allocated memory to grow with the number of mappings, unlike a statically allocated table. However, the OpenSSD firmware has only limited memory management features and uses a slow embedded ARM processor, which makes use of this data structure difficult.

Instead, our SSC FTL statically allocates a mapping table at device boot, and uses a lightweight hash function based on modulo operations and bit-wise rotations. To resolve conflicts between disk address that map to the same index, we use closed hashing with linear probing. While this raises the cost of conflicts, it greatly simplifies and shrinks the code.

Free-space management. FlashTier used hybrid address translation to reduce the size of the mapping table in the device. To reduce the cost of garbage collection, which must copy data to create empty erase blocks, FlashTier uses *silent eviction* to delete clean data rather than perform expensive copy operations.

In the prototype, we maintain a mapping of 4 KB pages, which reduces the cost of garbage collection because pages do not need to be coalesced into larger contiguous blocks. We found

this greatly improved performance for random workloads and reduced the cost of garbage collection.

We implement the silent eviction mechanism to reduce the number of copies during garbage collection. If a block has no dirty data, it can be discarded (evicted from the cache) rather than copied to a new location. Once the collector identifies a victim erase block, it walks through the pages comprising the block. If a page within the victim block is dirty, it uses regular garbage collection to copy the page, and otherwise discards the page contents.

We also implemented a garbage collector that uses hardware copyback support, which moves data within a bank, to improve performance. Similar to FlashTier, we reserve 7% of the device's capacity to accommodate bad blocks, metadata (e.g., for logging and checkpointing as described above), and to delay garbage collection. The FTL triggers garbage collection when the number of free blocks in a bank falls below a threshold. The collector selects a victim block based on the utilization of valid pages, which uses a hardware accelerator to compute the minimum utilization across all the erase blocks in the bank.

5.3.3 Lessons Learned

The implementation of address translation, free-space management, and consistency and durability mechanisms, raised several valuable lessons.

- Designs that rely on specific hardware capabilities or features, such as atomic writes, access to out-of-band area on flash, and dynamic memory management for device SRAM/DRAM, should consider the opportunity cost of using the feature, as other services within the SSD may have competing demands for it. For example, the OpenSSD flash controller stores ECC in the OOB area and prohibits its usage by the firmware. Similarly, the limited amount of SRAM requires a lean firmware image with statically linked libraries and necessitates the simpler data structure to store the address mapping.

- Many simulation studies use small erase block sizes, typically a few hundreds of KBs. In contrast, the OpenSSD platform and most commercial SSDs use larger erase block sizes from 1–20 MB to leverage the internal way and channel parallelism. This requires address

translation at a finer granularity, which makes it even more important to optimize the background operations such as garbage collection or metadata checkpointing whose cost is dependent on mapping and erase block sizes.

5.4 Development, Testing and Evaluation

In addition to the design challenges of implementing the SSC interfaces on OpenSSD, we were surprised by the difficulty of developing embedded firmware. These issues may be familiar to those who have written such code, but are a real challenge and significantly raise the complexity of testing a design with real hardware.

Development Framework. Writing FTL firmware entailed a substantial amount of device-driver programming. While the supplied firmware provided high-level routines for bad block management, event queues and several other hardware features, the flash operations themselves are accessed through device registers. The Flash Command Port (FCP) register set is used to initiate a flash operation and and the Bank Status Port (BSP) register set is used by the hardware controller to raise interrupts for the DMA controller and for bad block management. Thus, writing firmware required low-level interactions with the hardware.

Testing. Firmware development was complicated by the limited debugging facilities. The OpenSSD board has support for UART debugging, which can be used to transmit messages from the firmware over a serial interface to separate debugging machine. While useful for rare or unexpected events, the speed (115,200 bps) is too slow to log every request. In addition, they did little to debug system hangs. For example, when our firmware's metadata usage of SRAM exceeded the available limit of 96 KB, the host system would hang during BIOS initialization of the device. In another case the firmware buffer manager and queue manager deadlocked, hanging both the device and the host.

As a result, we largely relied on a *divide and conquer* approach to *split-the-FTL* functionality and separately test SATA communication between host and device from testing NAND communication between firmware and flash media. We modified the FTL to emulate commands using DRAM for storage rather than accessing flash. Similarly, we modify the firmware to

generate a synthetic sequence of flash commands to exercise the FTL and flash access functionality. Once two components work independently, we could test them together.

Performance Analysis. Finding bottlenecks in host-device communication and flash management is again a difficult process. Emulating NAND operations within device DRAM helped us to isolate the bottlenecks in host-device communication. For example, we accelerated our cache manager implementation for *exists* and *clean* commands, which only access device memory to test/set the dirty bitmap. We identified flash management bottlenecks by timing the firmware code using internal device timers, which are accurate to 3 µs. This was particularly useful in finding the optimal request size for flash read and write operations and to time checkpoint/recovery time for SSC metadata.

Implementation and Evaluation Timeline. We prototyped the SSC design on the OpenSSD platform in roughly 3 months. However, before and during that time, we spent about 2 months first understanding the development framework for both firmware programming and debugging. We spent about 2 months to find the performance bottlenecks in the reference FTL implementations, finding better ways to manage flash, and optimize the firmware to run at speeds close to raw flash. With debug assertions enabled to print messages over UART port, a single experiment expected to finish in a few minutes can take several minutes to hours, and sometimes even crash the full system. As a result, we spent about 1 month for making the firmware implementation robust for evaluation, and finally 1 month for evaluation and analysis.

5.5 Related Work

Our work builds on past work on prototyping flash devices and introducing new storage interfaces.

Hardware Prototypes. Research platforms for characterizing flash performance and reliability have been developed in the past [11, 13, 21, 56, 57]. In addition, there have been efforts on prototyping phase-change memory based prototypes [6, 15]. However, most of these

works have focused on understanding the architectural tradeoffs internal to flash SSDs and have used FPGA-based platforms and logic analyzers to measure individual raw flash chip performance characteristics, efficacy of ECC codes, and reverse-engineer FTL implementations. In addition, most FPGA-based prototypes built in the past have performed slower than commercial SSDs, and prohibit analyzing the cost and benefits of new SSD designs. Our prototyping efforts use OpenSSD with commodity SSD parts and have an internal flash organization and performance similar to commercial SSD. There are other projects creating open-source firmware for OpenSSD for research [108, 110] and educational purposes [19]. Furthermore, we investigated changes to the flash-device interface, while past work looks at internal FTL mechanisms.

New Flash Interfaces. In addition to FlashTier [99] and Nameless Writes [116], there have been commercial efforts on exposing new flash interfaces for file systems [50], caching [29, 48, 69] and key-value stores [30]. However, there is little known to the application developers about the customized communication channels used by the SSD vendors to implement new application-optimized interface. We focus on these challenges and propose solutions to overcome them.

While we re-use the existing SATA protocol to extend the SSD interface, another possibility is to bypass the storage stack and send commands directly to the device. For example, Fusion-io and the recent NVM Express specification [78] attach SSDs to the PCI express bus, which allows a driver to implement the block interface directly if wanted. Similarly, the Marvell DragonFly cache [67] bypasses SATA by using an RPC-like interface directly from a device driver, which simplifies integration and reduces the latency of communication.

5.6 Summary

Implementing novel SSD interfaces in a hardware prototype was a substantial effort, yet ultimately proved its value by providing a concrete demonstration of the performance benefits of FlashTier's SSC design.

Overall, we found the effort required to implement the two designs was comparable to

the effort needed to use a simulator or emulator. While we faced challenges integrating new commands into the operating system and firmware, with a simulator or emulator we would have struggled to accurately model realistic hardware and to ensure we appropriately handled concurrent operations. With real hardware, there is no need to validate the accuracy of models, and therefore, OpenSSD is a better environment to evaluate new SSD designs.

A major benefit of using OpenSSD is the ease of testing workloads. A design that only changes the internal workings of the device may be able to use trace-based simulation, but designs that change the interface have to handle new requests not found in existing traces. In contrast, working with OpenSSD allows us to run real application benchmarks and directly measure performance.

Working with a hardware prototype does raise additional challenges, most notably to add new commands to the storage stack. Here, our split-FTL approach may be promising. It leaves low-level flash operations in the device and runs the bulk of the FTL in the host OS. This approach may be best for designs with hardware requirements beyond the platform's capabilities, and for designs that radically change the device interface.

Finally, our prototyping efforts demonstrate that the ability to extend the interface to storage may ultimately be limited by how easily changes can be made to the OS storage stack. Research that proposes radical new interfaces to storage should consider how such a device would integrate into the existing software ecosystem. Introducing new commands and returning benign errors is difficult yet possible by tunneling them through native commands, and overloading the data path from the device to host. However, augmenting the control path with device upcalls requires significant changes to many OS layers, and thus may be better implemented through new communication channels.

6 CONCLUSIONS AND FUTURE WORK

Flash SSDs have become ubiquitous and can be used as an easy replacement to disk. In this dissertation, motivated by the differences in price, performance, and persistence of DRAM, SSDs and disks; we have designed new techniques to manage SSDs as an intermediate data tier.

In this section, we first summarize the contribution of this thesis, and then discuss future directions that can be further investigated.

6.1 Summary of Techniques

We presented two systems – FlashVM and FlashTier – that provide OS support to use SSDs as a memory tier between memory and disks. We now summarize the design techniques, which we re-visited or invented in the context of FlashVM and FlashTier.

Table 6.1 summarizes the different techniques designed as part of FlashVM and FlashTier, which improve performance, and reliability, reduce crash-consistency cost, and save host memory. We also list if and how these techniques can benefit systems using emerging SCM technologies, and disks or disk arrays instead of flash SSDs.

FlashVM techniques modify the paging system on code paths affected by the performance differences between flash and disk: on the read path during a page fault, and on the write path when pages are evicted from memory. On the read path, FlashVM leverages the low seek time on flash to prefetch more useful pages through stride prefetching to minimize memory pollution. This results in a reduction in the number of page faults and improves the application execution time. Virtual memory backed with SCM would also provide fast random access for reads, and we project similar benefits for stride prefetching. However, disks would suffer from the seek overheads for stride prefetching.

On the write path, FlashVM throttles the page write-back rate at a finer granularity than Linux. This allows better congestion control of paging traffic to flash and improved page fault latencies for various application workloads. Similarly, FlashVM pre-cleans a set of dirty pages

System	**T**echnique	**F**lash	**S**CM	**D**isk
Virtual Memory	Stride Prefetching	Perf	Perf	NA
	Page Pre-cleaning	Perf	Perf	Perf
	Page Clustering	Perf+Rel	Perf	Perf
	Page Scanning	Perf	Perf	Perf
	I/O Scheduling	NA	NA	Perf
	Page Sampling	Rel+Perf	Rel	NA
	Page Sharing	Rel+Perf	Rel+Perf	Perf
	Merged Discard	Rel+Perf	Rel	NA
	Dummy Discard	Rel+Perf	Rel	NA
Caching	Unified Address Space	Perf+Mem+Const	Mem+Const	NA
	Sparse Mapping	Perf+Mem	Perf+Mem	NA
	Silent Eviction	Perf+Rel	Perf+Rel	Perf
	Cache-aware Interface	Perf+Const	Perf+Const	Rel
	Transactional Logging	Perf+Const	Perf+Const	Perf+Const
SSC Prototype	Merge Buffer	Perf+Rel	Perf+Rel	Perf
	Read Buffer	Perf	Perf	Perf
	Tunneling New Commands	Const	NA	Const
	Hardware Acceleration	Perf+Const	NA	Perf
	Overloading Data Path	Perf+Const	NA	Const
	Split-FTL Testing	Debugging+Analysis		

Table 6.1: **S**ystems and Techniques: Benefits of techniques implemented as part of FlashVM, FlashTier and SSC, for systems using Flash SSDs, Storage-Class Memory and Disks. (Perf: Performance, Rel: Reliability, Mem: Memory Overhead, Const: Consistency Cost, NA: zero or negative benefits).

and writes them back as a sequential cluster. These optimizations along the write path will benefit both SCM and disk-based systems because SCM also possess asymmetrical read/write performance (see Table 2.1). Page scanning is the only write optimization, which needs to adapt to access times for disk-backed VM differently.

The write path also affects the reliability of virtual memory. Flash memory suffers from wear out, in that a single block of storage can only be written a finite number of times. FlashVM uses zero-page sharing to avoid writing empty pages and uses page sampling, which probabilistically skips over dirty pages to prioritize the replacement of clean pages. Both techniques reduce the number of page writes to the flash device, resulting in improved reliability for FlashVM. Page writes to SCM are faster than flash, which results in more write-backs to SCM. However, SCM also suffers from limited write endurance, therefore both page sharing

and sampling would benefit SCM. Reducing the number of writes to disk-backed VM is mainly a performance benefit.

The third focus of FlashVM is efficient garbage collection, which affects both reliability and performance. Modern SSDs provide a discard command (also called trim) for the OS to notify the device when blocks no longer contain valid data [102]. FlashVM presents the first comprehensive description of the semantics, usage and performance characteristics of the discard command on a real SSD. In addition, FlashVM proposes two different techniques, merged and dummy discards, to optimize the use of the discard command for online garbage collection of free VM page clusters on the swap device. SCM as memory extension can also benefit form hints like discard for reducing the number of re-writes during static wear-leveling. However, such notifications may not benefit disk-backed VM in the absence of any garbage collection requirements.

FlashTier techniques customize the device interface and OS mechanisms for using flash SSDs as a cache in front of disks.

FlashTier exploits the three features of caching workloads to improve over SSD-based caches. First, FlashTier provides a unified address space that allows data to be written to the SSC at its disk address. This removes the need for a separate table mapping disk addresses to SSD addresses. In addition, an SSC uses internal data structures tuned for large, sparse address spaces to maintain the mapping of block number to physical location in flash. Using SCM as flash or disk caching tier would introduce another level in the memory hierarchy, and unified address spaces across different data tiers would provide similar benefits. However, unifying address space for multiple disks in an array may pose additional challenges during array reconstruction or recovery.

Second, FlashTier provides cache consistency guarantees to ensure correctness following a power failure or system crash. It provides separate guarantees for clean and dirty data to support both write-through and write-back caching. In both cases, it guarantees that stale data will never be returned. Furthermore, FlashTier introduces new operations in the device interface to manage cache contents and direct eviction policies. FlashTier ensures that

internal SSC metadata is always persistent and recoverable after a crash using a lightweight transaction mechanism, allowing cache contents to be used after a failure. Classification of data into clean and dirty pages is equally important for SCM-backed caches to enable better protection and wear management of dirty data. It can also be useful to preferentially stripe dirty data across different disks in an array for more reliability.

Finally, FlashTier leverages its status as a cache to reduce the cost of free space management. Unlike a storage device, which promises to never lose data, a cache can evict blocks when space is needed. For example, flash must be erased before being written, requiring a garbage collection step to create free blocks. An SSD must copy live data from blocks before erasing them, requiring additional space for live data and time to write the data. In contrast, an SSC may instead silently evict the data, freeing more space faster. For SCM-backed caches, silent eviction can not only reduce the penalty for re-writing clean pages for wear management, but we can also preferentially evict clean pages from DRAM, and turn down some DIMMs at the cost of reading evicted pages back from SCM on a re-reference. Silent eviction can also be used to reduce the costly read-modify-write cycles for data mirrored across different disks in an array.

SSC prototype led us to learn various lessons as part of re-implementing FlashTier's SSC design on a hardware platform with more realistic constraints. For example, we found that with large block sizes and increased parallelism in modern flash devices, merging and buffering to align requests within the device memory is imperative to fully saturate the internal bandwidth provide by flash. Similarly, we reduce the number of data copies by initiating DMA transfers to host from the read buffer for a page without waiting to read it as part of new full block read. For SCM-backed memory across the network, similar RDMA transfers can minimize the number of data copies and significantly improve read latencies. Read and write buffering is useful to align I/O requests to stripe size in the context of disk arrays.

Second, we also implement OS storage stack extensions to introduce new command interfaces for the SSC prototype. We tunneled new forward commands from the OS to the device by adding new subtypes to native read and write commands. However, this necessitated

the introduction of new rules at I/O scheduler and firmware to ensure well-defined consistency semantics for new commands. Reverse device responses are much difficult to implement, for example, we have to overload benign errors returned from the SSC on the read response data path. Finally, within the SSC, we used the fast path for servicing new commands through the hardware queue by masking command subtypes in the length field of native commands. SCM-backed caches can be directly addressed without going through the block layer and would not require extensions to the storage software stack. However, introducing new extensions to SCM load/store would require additional processor primitives. For disk arrays, new commands would require similar changes through the array device drivers.

Finally, over the course of prototyping SSC on the OpenSSD platform, we learned how to split the FTL functionality across host and device to debug and analyze the performance of different mechanisms for flash management firmware. For example, we debug the new flash interface in the OS separately from the firmware changes within the device. This split-FTL functionality is most useful for identifying a clean separation of responsibility between the host and device. SCM and disk arrays can also leverage from split designs for debugging, analysis and block management.

6.2 Lessons Learned

In this section, we present a list of major lessons we learned while working on this dissertation.

Software-hardware co-design is key for new memory technologies. Our experience building new software abstractions and interfaces to flash SSDs made us learn that both software and hardware must evolve together. Flash SSDs are increasingly becoming advanced with support for hardware accelerators [70], large device memory sizes [44], increased processing power [44], more internal parallelism [31], deduplication and compression [65], and robust error correction mechanisms [117]. The OS and sofware can be re-designed to leverage the functionality provided by advancements in hardware without duplicating it in software.

However, software abstractions need more control through new interfaces for better visibility on how operations are scheduled on hardware. For example, the existing block interface of

SSDs meant for emulating disks is too narrow to provide any control to software. Operations with non-determistic cost such as background space and wear management, erase cycles, and deduplication provide un-predictable latencies and throughput to executing applications. In this thesis, we have taken the first step towards exposing custom hardware interfaces to OS and applications, and we expect more potential in this direction.

Similarly, we learned while prototyping the SSC on a real hardware platform that we should consider all layers of the OS for supporting new memory technologies. Past research has largely focused on the end-points of the storage stack, file systems and the storage subsystem. In this thesis, we found that the interface between these end-points is actually composed of several layers, including the block layer, I/O scheduler, device drivers, firmware, and hardware queues. These intermediate layers are composed of several hundreds of thousands of lines of code and need to be re-visited altogether for modern hardware technologies.

System designs must address fundamental flash media properties, not its idiosyncrasies. Flash and non-volatile memory technologies are rapidly evolving in terms of performance and reliability characteristics. System designs must focus and embrace fundamental properties of flash medium, such as fast random access, costly erase operations, large page and erase block sizes, flash wear-out and endurance, and internal device parallelism.

However, system designs that optimize for flash idiosyncrasies, such as low random write performance, can be specific to a generation of the technology, and has been addressed over the last few years, with support for improved log-structuring, over-provisioning, and in-device buffering. In summary, research focusing on the fundamental properties inherent to the flash medium presents opportunities for long-lasting impact on the design of new software abstractions.

6.3 Future Work

We outline future directions in the context of this dissertation that could be further investigated.

Paging for Fast SSDs and Fast Networks. FlashVM addresses the challenges for paging to commodity flash SSDs, which are still an order of magnitude slower than memory. As

a result, software latency is only a small fraction of the total page fault latency. However, with faster SSDs and SCM technologies with latencies close to memory, the overheads of page scanning and context switching can be a large fraction of page fault latencies [97]. This makes synchronous or blocking page faults more attractive for future platforms, which lack any overhead for switching process context or for interrupt handling.

Similarly, with the advent of fast optical networks [77] and Infiniband [43], network latencies are becoming dominated by overhead for page copies from fast remote storage (SCM or PCI-e flash [1]) to remote memory to local memory. In those scenarios, new remote DMA mechanisms or mechanisms to directly map remote PCI-e bus into local memory, are worth investigating for fast paging over network.

Policies for Caching and Tiering. FlashTier uses a cost-benefit based policy to select a block for silent eviction within the SSC. The cost is determined by the number of valid pages in the block being evicted and benefit is determined by the last time it was last modified measuring its age. This policy is inspired by LRU cache replacement and segment cleaning in log-structured file system [95]. An interesting question is how different cache replacement policies such as adaptive replacement cache [35] would impact cache performance. Similarly, it is worth investigating how more information supplied from OS on both read/write usage, and semantics of blocks would impact performance and reliability of SSC silent eviction.

Unlike caching, tiering allows SSDs to be used in the same address space as disks and adds to the total storage capacity. Tiering is more effective for storage controllers, where a set of SSDs can be introduced in a hybrid disk array. However, tiering would increase heterogeneity compared to a caching-based solution both in terms of performance and reliability characteristics of the entire array. In addition, new tiering policies need to be investigated to automatically identify and migrate hot or popular data into SSDs after invalidating it in the disk tier to maintain high performance and well-defined consistency semantics.

Caching for Multiple Devices. In FlashTier, we unify the address space across the host memory and SSC device memory. This results in reduced host memory usage and reduction in cost of consistency for maintaining only level of mapping persistent. However, this limits

the disk capacity addressed by the size of the translation table stored in SSC device memory. Similarly, if we add more SSCs for caching, we need to distribute the mapping across all of them, sometimes replicating an entry across different SSCs corresponding to the disk blocks cached on multiple SSC devices. In summary, it is worth investigating how we can unify address space in the presence of arrays of SSC devices or disks. One solution is to store the unified address map within the host memory instead of device memory. However, this increases the overhead for persisting the mapping by flushing metadata updates from host to SSC.

Isolation for Multiple Applications. In FlashTier, we focus on boosting the performance of a stream of block I/O requests regardless of the applications. Although we provide support for selective caching to bypass the cache for sequential I/O flows, we can not yet provide performance isolation across different workloads using the cache. The problem of performance isolation for caches and storage is not new and has been investigated in the past [111]. However, the lack of predictability with flash makes it challenging for the OS or software to provide performance isolation or accountability to different workloads. The un-predictability stems from various factors including background operations, queuing of erase/write before read operations, multiple channels and banks in SSDs, and increased flash wear with prolonged use. It is worth investigating to find a mechanism, which can provide more control to software for providing performance isolation.

6.4 Closing words

The availability of high-speed flash SSDs has introduced a new data tier in the memory hierarchy. In this thesis, we have taken the first steps to build systems that unleash the full power of flash SSDs as extended memory and caches between memory and disks. Future directions outlined above pave the way for further work in designing advanced systems for software management of flash and emerging storage-class memory technologies.

BIBLIOGRAPHY

[1] Fusion-IO ioXtreme PCI-e SSD Datasheet. http://www.fusionio.com/ioxtreme/PDFs/ioXtremeDS_v.9.pdf.

[2] Micron Omneo P8P 128-Mbit Parallel PCM. http://www.micron.com/products/phase-change-memory/parallel-pcm.

[3] G. Aayush, K. Youngjae, and U. Bhuvan. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS*, 2009.

[4] N. Agrawal, L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Emulating Goliath storage systems with David. In *FAST*, 2011.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX*, 2008.

[6] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A protoype phase-change memory storage array. In *HotStorage*, 2011.

[7] Amazon. High IO EC2 SSD Instances. http://aws.amazon.com/about-aws/whats-new/2012/07/18/announcing-high-io-instances-for-amazon-ec2.

[8] T. Archer. MSDN Blog: Microsoft ReadyBoost. http://blogs.msdn.com/tomarcher/archive/2006/06/02/615199.aspx.

[9] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: FacebookâĽ™s photo storage. In *OSDI*, 2010.

[10] T. Bisson. Reducing hybrid disk write latency with flash-backed io requests. In *MASCOTS*, 2007.

[11] S. Boboila and P. Desnoyers. Write endurance in flash drives: Measurements and analysis. In *FAST*, 2010.

[12] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *SOSP*, 1997.

[13] T. Bunker, M. Wei, and S. Swanson. Ming II: A flexible platform for nand flash-based research. In *UCSD TR CS2012-0978*, 2012.

[14] S. Byan, J. Lentini, L. Pabon, C. Small, and M. W. Storer. Mercury: host-side flash caching for the datacenter. In *FAST Poster*, 2011.

[15] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *IEEE Micro*, 2010.

[16] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA*, 2011.

[17] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *FAST*, 2011.

[18] Complex. Bubble Memory: 10 Technologies that were supposed to Blow Up but Never Did. `http://www.complex.com/tech/2012/10/10-technologies-that-were-supposed-to-blow-but-never-did/bubble-memory`.

[19] Computer Systems Laboratory, SKKU. Embedded Systems Design Class. `http://csl.skku.edu/ICE3028S12/Overview`.

[20] J. Corbet. Barriers and journaling filesystems, May 2008. `http://lwn.net/Articles/283161/`.

[21] J. D. Davis and L. Zhang. FRP: a nonvolatile memory research platform targeting nand flash. In *Workshop on Integrating Solid-state Memory into the Storage Hierarchy, ASPLOS*, 2009.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[23] Dotnetperls.com. Memory benchmark for Web Browsers., December 2009. `http://dotnetperls.com/chrome-memory`.

[24] S. Doyle and A. Narayan. Enterprise solid state drive endurance. In *Intel IDF*, 2010.

[25] DRAMeXchange.com. Mlc nand flash price quote, dec 2009. `http://dramexchange.com/flash`.

[26] DRAMeXchange.com. Dram and nand flash spot prices, apr 2013. `http://dramexchange.com`.

[27] EMC. Fully Automated Storage Tiering (FAST) Cache. `http://www.emc.com/about/glossary/fast-cache.htm`.

[28] Facebook Inc. Facebook FlashCache. `https://github.com/facebook/flashcache`.

[29] Fusion-io Inc. directCache. `http://www.fusionio.com/data-sheets/directcache`.

[30] Fusion-io Inc. ioMemory Application SDK. `http://www.fusionio.com/products/iomemorysdk`.

[31] FusionIO. ioDrive. `www.fusionio.com/products/iodrive`.

[32] FusionIO. ioTurbine: Accelerated Virtualized Application Performance. `http://www.fusionio.com/systems/ioturbine`.

[33] Gear6. Scalable hybrid memcached solutions. `http://www.gear6.com`.

[34] Google Inc. Google Sparse Hash. `http://goog-sparsehash.sourceforge.net`.

[35] B. Gregg. Sun Blog: Solaris L2ARC Cache. `http://blogs.sun.com/brendan/entry/test`.

[36] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, 2011.

[37] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In *SOSP*, pages 293–306, Oct. 2007.

[38] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *FAST*, 2011.

[39] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *OSDI*, 2008.

[40] P. Helland, H. Sammer, J. Lyon, R. Carr, and P. Garrett. Group commit timers and high-volume transaction systems. In *Tandem TR 88.1*, 1988.

[41] Y. Huai. Spin-transfer torque mram (STT-MRAM): Challenges and prospects. *AAPPS Bulletin*, 18(6):33–40, Dec. 2008.

[42] H. Iizuka and F. Masuoka. Semiconductor memory device and method for manufacturing the same. In *US Patent 4531203A*, 1980. `http://www.google.com/patents/US4531203`.

[43] Intel. Infiniband Architecture: The high-performance Interconnect. `http://www.intel.com/content/www/us/en/data-center/data-center-management/infiniband-architecture-general.html`.

[44] Intel. SSD DC S3700 Controller with 1 GB DRAM. `http://www.anandtech.com/print/6432`.

[45] Intel. X-25 mainstream ssd datasheet. `http://tinyurl.com/qlu425`.

[46] Intel. Understanding the flash translation layer (ftl) specification, Dec. 1998. Application Note AP-684.

[47] Intel Corp. Intel 300 series SSD. `http://ark.intel.com/products/family/56542/Intel-SSD-300-Family`.

[48] Intel Corp. Intel Smart Response Technology. `http://download.intel.com/design/flash/nand/325554.pdf`, 2011.

[49] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous IO. In *SOSP*, 2001.

[50] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: a file system for virtualized flash storage. In *FAST*, 2010.

[51] T. Kgil and T. N. Mudge. Flashcache: A nand flash memory file cache for low power web servers. In *CASES*, 2006.

[52] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. FlashSim: A simulator for nand flash-based solid-state drives. *Advances in System Simulation, International Conference on*, 0:125–131, 2009.

[53] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. In *FAST*, 2010.

[54] M. H. Kryder and C. S. Kim. After hard drives – what comes next? In *IEEE Transactions on Magnetics*, 2009.

[55] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[56] S. Lee, K. Fleming, J. Park, K. Ha, A. M. Caulfield, S. Swanson, Arvind, , and J. Kim. BlueSSD: An open platform for cross-layer experiments for nand flash-based ssds. In *Workshop on Architectural Research Prototyping*, 2010.

[57] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. FlexFS: a flexible flash file system for mlc nand flash memory. In *Usenix ATC*, 2009.

[58] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst*, 6(3), July 2007.

[59] H. M. Levy and P. H. Lipman. Virtual memory management in the VAX/VMS operating system. *Computer*, 15(3):35–41, 1982.

[60] H.-L. Li, C.-L. Yang, and H.-W. Tseng. Energy-aware flash memory management in virtual memory system. In *IEEE Transactions on VLSI systems*, 2008.

[61] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *SOSP*, 2011.

[62] Linux-Documentation. SCSI Generic Driver Interface. `http://tldp.org/HOWTO/SCSI-Generic-HOWTO`.

[63] M. Lord. Author, Linux IDE Subsystem, Personal Communication. December, 2009.

[64] M. Lord. hdparm 9.27: get/set hard disk parameters. `http://linux.die.net/man/8/hdparm`.

[65] L. Magazine. On-the-fly Data Compression for SSDs. `http://www.linux-mag.com/id/7869`.

[66] B. Martin. Inspecting disk io performance with fio. `http://www.linuxtoday.com/storage/20080410005260SHL`, Apr. 2008.

[67] Marvell Corp. Dragonfly platform family. `http://www.marvell.com/storage/dragonfly/`, 2012.

[68] memcached. High-performance Memory Object Cache. `http://www.danga.com/memcached`.

[69] M. Mesnier, J. B. Akers, F. Chen, and T. Luo. Differentiated storage services. In *SOSP*, 2011.

[70] Microsoft. Windows 8 eDrive Standard. http://technet.microsoft.com/en-us/library/hh831627.aspx.

[71] N. Mielke, T. Marquart, J. N. W. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, E. Nevill, and L.R. Bit error rate in nand flash memories. In *IEEE IRPS*, 2008.

[72] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for nvm+dram hybrid main memory. In *HotOS*, 2009.

[73] MSDN Blog. Trim Support for Windows 7. http://blogs.msdn.com/e7/archive/2009/05/05/support-and-q-a-for-solid-state-drives-and.aspx.

[74] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-loading: Practical power management for enterprise storage. In *FAST*, 2008.

[75] D. Nellans, M. Zappe, J. Axboe, and D. Flynn. ptrim() + exists(): Exposing new FTL primitives to applications. In *NVMW*, 2011.

[76] NetApp Inc. Flash Cache for Enterprise. http://www.netapp.com/us/products/storage-systems/flash-cache.

[77] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *OSDI*, 2012.

[78] NVM Express. Nvm express revision 1.0b. http://www.nvmexpress.org/index.php/download_file/view/42/1/, July 2011.

[79] Objective-Analysis.com. White Paper: PCM becomes a reality., Aug. 2009. http://www.objective-analysis.com/uploads/2009-08-03_Objective_Analysis_PCM_White_Paper.pdf.

[80] OCZ. OCZ Synapse Cache SSD. http://www.ocztechnology.com/ocz-synapse-cache-sata-iii-2-5-ssd.html.

[81] OCZ Technologies. Vertex 3 SSD. http://www.ocztechnology.com/ocz-vertex-3-sata-iii-2-5-ssd.html.

[82] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, 2012.

[83] Oracle. SPARC Enterprise M9000 Server. http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/m-series/m9000/overview/index.html.

[84] Oracle Corp. Oracle Database Smart Flash Cache. http://www.oracle.com/technetwork/articles/systems-hardware-architecture/oracle-db-smart-flash-cache-175588.pdf.

[85] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D.K.Panda. Beyond block i/o: Rethinking traditional storage primitives. In *HPCA*, pages 301 –311, feb. 2011.

[86] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min. Compiler-assisted demand paging for embedded systems with flash memory. In *EMSOFT*, 2004.

[87] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee. CFLRU: A replacement algorithm for flash memory. In *CASES*, 2006.

[88] S. Parkin, X. Jiang, C. Kaiser, A. Panchula, K. Roche, and M. Samant. Magnetically engineered spintronic sensors and memory. *Proceedings of the IEEE*, 91(5):661 – 680, May 2003.

[89] M. Polte, J. Simsa, and G. Gibson. Enabling enterprise solid state disks performance. In *WISH*, 2009.

[90] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *SOSP*, pages 206–220, 2005.

[91] V. Prabhakaran, T. Rodeheffer, and L. Zhou. Transactional flash. In *OSDI*, 2008.

[92] A. Rajimwale, V. Prabhakaran, and J. D. Davis. Block management in solid-state devices. In *USENIX*, 2009.

[93] Redhat Inc. JFFS2: The Journalling Flash File System, version 2, 2003. `http://sources.redhat.com/jffs2`.

[94] D. Roberts, T. Kgil, and T. Mudge. Integrating NAND flash devices onto servers. *CACM*, 52(4):98–106, Apr. 2009.

[95] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992.

[96] Ryan Mack. Building Facebook Timeline: Scaling up to hold your life story. `http://www.facebook.com/note.php?note_id=10150468255628920`.

[97] M. Saxena and M. M. Swift. FlashVM: Revisiting the Virtual Memory Hierarchy. In *HotOS-XII*, 2009.

[98] M. Saxena and M. M. Swift. FlashVM: Virtual Memory Management on Flash. In *Usenix ATC*, 2010.

[99] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A lightweight, consistent and durable storage cache. In *EuroSys*, 2012.

[100] M. Saxena, Y. Zhang, M. M. Swift, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Getting real: Lessons in transitioning research simulations into hardware systems. In *FAST*, 2013.

[101] Seagate. NAND Flash Supply Market, 2012. `http://www.seagate.com/em/en/point-of-view/nand-flash-supply-market-master-pov`.

[102] F. Shu and N. Obr. Data Set Management Commands Proposal for ATA8-ACS2, 2007. `http://t13.org/Documents/UploadedDocuments/docs2008/e07154r6-Data_Set_Management_Proposal_for_ATA-ACS2.doc`.

[103] Spin. LTL Model Checker, Bell Labs. http://spinroot.com.

[104] STEC Inc. EnhanceIO. https://github.com/stec-inc/EnhanceIO.

[105] StreetPrices.com. Disk drive price quote, Dec. 2009. http://www.streetprices.com.

[106] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, 2008.

[107] The OpenSSD Project. Indilinx Jasmine Platform. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[108] The OpenSSD Project. Participating Institutes. http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform.

[109] Twitter. FatCache: memcache on SSDs. https://github.com/twitter/fatcache.

[110] VLDB Lab. SKKU University, Korea. http://ldb.skku.ac.kr.

[111] M. Wachs, M. A. el malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, 2007.

[112] Wikipedia. Millipede Memory: MEMS. http://en.wikipedia.org/wiki/Millipede_memory.

[113] T. M. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Usenix ATC*, 2002.

[114] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *ASPLOS-VI*, 1994.

[115] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-all replacement for a multilevel cache. In *FAST*, 2007.

[116] Y. Zhang, L. P. Arulraj, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, 2012.

[117] K. Zhao, W. Zhao, H. Sun, T. Zhang, X. Zhang, and N. Zheng. Ldpc-in-ssd: Making advanced error correction codes work effectively in solid state drives. In *FAST*, 2013.