Design and Prototype of a Solid-State Cache

MOHIT SAXENA and MICHAEL M. SWIFT, University of Wisconsin-Madison

The availability of high-speed solid-state storage has introduced a new tier into the storage hierarchy. Low-latency and high-IOPS solid-state drives (SSDs) cache data in front of high-capacity disks. However, most existing SSDs are designed to be a drop-in disk replacement, and hence are mismatched for use as a cache.

This article describes *FlashTier*, a system architecture built upon a *solid-state cache* (SSC), which is a flash device with an interface designed for caching. Management software at the operating system block layer directs caching. The FlashTier design addresses three limitations of using traditional SSDs for caching. First, FlashTier provides a unified logical address space to reduce the cost of cache block management within both the OS and the SSD. Second, FlashTier provides a new SSC block interface to enable a warm cache with consistent data after a crash. Finally, FlashTier leverages cache behavior to silently evict data blocks during garbage collection to improve performance of the SSC.

We first implement an SSC simulator and a cache manager in Linux to perform an in-depth evaluation and analysis of FlashTier's design techniques. Next, we develop a prototype of SSC on the OpenSSD Jasmine hardware platform to investigate the benefits and practicality of FlashTier design. Our prototyping experiences provide insights applicable to managing modern flash hardware, implementing other SSD prototypes and new OS storage stack interface extensions.

Overall, we find that FlashTier improves cache performance by up to 168% 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 flash interface.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]; D.4.2 [**Operating Systems**]: Storage Management

General Terms: Design, Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Solid-state cache, device interface, consistency, durability, prototype

ACM Reference Format:
Mohit Saxena and Michael M. Swift. 2014. Design and prototype of a solid-state cache. ACM Trans. Storage 10, 3, Article 10 (July 2014), 34 pages.
DOI: http://dx.doi.org/10.1145/2629491

1. INTRODUCTION

Solid-state drives (SSDs) composed of multiple flash memory chips are often deployed as a cache in front of cheap and slow disks [Kgil and Mudge 2006; EMC 2013; Zhang et al. 2013; Koller et al. 2013]. 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 [Facebook, Inc. 2013; STEC, Inc. 2012; Mack 2012; Oracle Corp. 2012]. 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 [Intel Corp. 2011; OCZ 2012; Fusion-io, Inc. 2013a] 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

Authers' address: Mohit Saxena and Michael M. Swift, University of Wisconsin-Madison.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2014 ACM 1553-3077/2014/07-ART10 $15.00
DOI: http://dx.doi.org/10.1145/2629491

replacement [Agrawal et al. 2008; Prabhakaran et al. 2008; Wu and Zwaenepoel 1994]. 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 article describes *FlashTier* [Saxena et al. 2012, 2013], 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.

FlashTier exploits the three features of caching workloads to improve upon 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.

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, allowing cache contents to be used following a failure.

Finally, FlashTier leverages its status as a cache to reduce the cost of garbage collection. 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 rewrite the data. In contrast, an SSC may instead *silently evict* the data, freeing more space faster.

Initially, we implemented an SSC simulator and a cache manager for Linux and evaluate FlashTier on four different storage traces [Saxena et al. 2012]. We perform an in-depth analysis by measuring the cost and benefits of each of our design techniques. Our simulation results show significant potential for achieving performance and reliability benefits of SSC design compared to SSD caches.

Next, we sought to validate the benefits of SSC design by implementing it as a prototype on the OpenSSD Jasmine hardware platform [Saxena et al. 2013]. The OpenSSD evaluation board is composed of commodity SSD parts, including a commercial flash controller, and supports standard storage interfaces (SATA). It allows the firmware to be completely replaced, and therefore enables the introduction of new commands or changes to existing commands in addition to changes to the flash translation layer

Table I. Device Attributes

Device	Access Latency		Capacity	Price	Endurance	Power
	Read	Write	Bytes	$/GB	Erases	Watts
DRAM	50 ns	50 ns	8 GB	$5	∞	4–5 W/DIMM
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

Price, performance, endurance, and power of DRAM, NAND Flash SSDs and Disk. (MB: megabyte, GB: gigabyte, TB: terabyte, as of April 2013).

(FTL) algorithms. As a real storage device with performance comparable to commercial SSDs, our experiences from prototyping on the OpenSSD platform led us to learn several lessons on the practicality of SSC design and new flash interfaces.

Overall, our results show the following.

—FlashTier reduces total memory usage by more than 60% compared to existing systems using an SSD cache.
—FlashTier's free space management improves performance by up to 168% compared to consumer-grade SSDs and by up to 52% compared to high-end SSDs. It also improves flash lifetime by up to 60% for write-intensive workloads.
—After a crash, FlashTier can recover a 100-GB cache in less than 2.4 seconds, much faster than existing systems providing consistency on an SSD cache.

The remainder of the article is structured as follows. Section 2 describes our caching workload characteristics, motivates FlashTier and the need for our prototyping exercise. Section 3 presents an overview of FlashTier design, followed by a detailed description in Section 4. We present implementation in simulation and on prototype in Section 5. Next, we describe a set of lessons from our prototyping experiences in Section 6. Finally, we evaluate FlashTier design techniques in Section 7 and conclude with a description of related work in Section 8.

2. 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 I). 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 [OCZ 2012; NetApp, Inc. 2013; Mack 2012; Saxena and Swift 2010; Kgil and Mudge 2006].

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.

2.1. Why FlashTier?

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

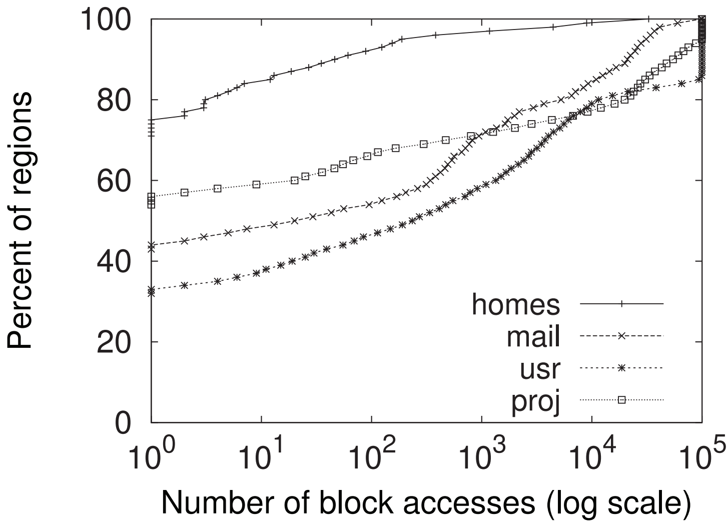

Fig. 1. Logical Block Addresses Distribution. The distribution of unique block accesses across 100,000 4 KB block regions of the disk address space.

sets of production systems downstream of an active page cache over 1 to 3 week periods [Koller and Rangaswami 2010; Narayanan et al. 2008]. These systems have different I/O workloads that consist of a file server (*homes* workload), an e-mail server (*mail* workload) and file servers from a small enterprise data center hosting user home and project directories (*usr* and *proj*). Table V (see Section 7.1) summarizes the workload statistics. Trends observed across all these workloads directly motivate our design for FlashTier.

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 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 have less than 1% of their blocks referenced, and only 25% of the regions have 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, making 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, and discard all cached data after a crash [Byan et al. 2011; Gregg 2008].

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 [Corbet 2008]. 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 [Doyle and Narayan 2010].

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 four times greater than 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 to maximize cache hit rates 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 [Intel 1998; Aayush et al. 2009].

2.2. Why Prototype SSC?

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 [Mesnier et al. 2011; Oh et al. 2012]), optimizing their internal algorithms (e.g., garbage collection [Chen et al. 2011; Aayush et al. 2009; Gupta et al. 2011]), and extending their interface (e.g., caching operations [Saxena et al. 2012]). However, most research looking at internal SSD mechanisms relies on simulation rather than direct experimentation [Agrawal et al. 2008; Oh et al. 2012; Saxena et al. 2012; Zhang et al. 2012].

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 [Intel Corp. 2011; OCZ 2012; Fusion-io, Inc. 2013a, 2013b], 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 provide a more controlled environment to analyze and evaluate the impact of new flash management algorithms. In addition, new SSD and interface designs are easier to implement with simulators than prototypes. However, simulators suffer

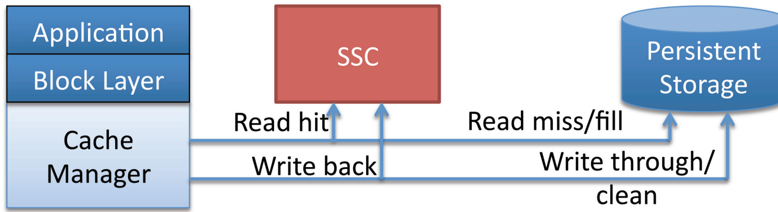

Fig. 2. FlashTier Data Path. A cache manager forwards block read/write requests to disk and solid-state cache.

from two major sources of inaccuracy. First, they are limited by the quality of performance models, which may miss important real-world effects, such as the complex interaction of I/O requests across multiple parallel flash banks in an SSD. Second, simulators often simplify systems and may leave out important components, such as the software stack used to access an SSD.

As a result, we first implement an SSC simulator [Saxena et al. 2012] to evaluate the costs and benefits of SSC design techniques. Next, we develop the SSC prototype [Saxena et al. 2013] to validate the performance claims of simulation and verify the practicality of new SSC interface and mechanisms.

3. 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 2 shows the flow of read and write requests from the application to SSC and disk-storage tiers from the cache manager.

3.1. 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. This is useful when there is little benefit to caching writes, or 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 invalidate 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.

3.2. 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 a 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.

3.3. 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 previously, 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 a fixed capacity for overprovisioned blocks to make free space available. Most SSDs reserve 5%–20% of their capacity to create free erased blocks to accept writes [Intel Corp. 2012; Doyle and Narayan 2010; OCZ Technologies 2012]. However, because an SSC can use a variable capacity for overprovisioning, it can flexibly dedicate space either to data, to reduce miss rates, or to the log, to accept writes.

3.4. 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 [Byan et al. 2011; Gregg 2008].

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.

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 [Prabhakaran et al. 2008; Ouyang et al. 2011].

4. SYSTEM DESIGN

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

—*Address space management* unifies address space translation and block state between the OS and SSC and optimizes for sparseness of cached blocks.
—*Free space management* improves cache write performance by silently evicting data rather than copying it within the SSC.
—*Consistent interface* provides consistent reads after cache writes and eviction and makes 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.

4.1. 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 [Josephson et al. 2010], 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 [Google, Inc. 2012]. 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 t buckets divided into t/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 i is set to 1 if and only if bucket i is non-empty. A lookup for bucket i calculates the value location from the number of 1s in the bitmap before location 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 [Google, Inc. 2012]. The runtime of all operations on the hash map is bounded by the constant M, and typically there are no more than 4 to 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 [Lee et al. 2007; Intel 1998]. 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 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 [Chen et al. 2011] that can be written at the same time as data. The OOB area is also used to store error-correction codes for flash wear management in most modern SSDs. 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.

4.2. 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 [Nellans et al. 2011]. 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.

Table II. The Solid-State Cache Interface

Name	Function
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.

4.2.1. Interface. FlashTier's interface consists of six operations, as listed in Table II. 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 to 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 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.

4.2.2. 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, the 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 [OpenSSD Project Website 2013].

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 [Helland et al. 1988] 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 [Ouyang et al. 2011] 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 [OpenSSD Project Website 2013] (see Section 6), 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 the OpenSSD platform, we defer checkpoints until requested by host using a *flush* operation, which allows the cache manager in the OS or application define ordering 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. 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 [Intel 1998; Aayush et al. 2009].

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 here. 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 [Rosenblum and Ousterhout 1992].

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* (solid-state cache) 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* (solid-state cache with variable log space) 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 variable 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 [Aayush et al. 2009], which convert a sequentially written log block into a data block without copying data.

4.4. Cache Manager

The cache manager is based on Facebook's FlashCache for Linux [Facebook, Inc. 2013]. 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.

5. IMPLEMENTATION

The implementation of FlashTier entails three components: the cache manager, an SSC timing simulator, and the OpenSSD hardware prototype. The cache manager is a Linux kernel module (kernel 2.6.33), and the simulator models the time for the completion of each request. We customize the OpenSSD reference FTL firmware to implement and validate the benefits of the new flash interface and internal mechanisms of the SSC.

We base the cache manager on Facebook's FlashCache [Facebook, Inc. 2013]. 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.

5.1. SSC Simulator Implementation

We base the SSC simulator on FlashSim [Kim et al. 2009]. 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 to 20% for SSC-V.

We implemented our own FTL that is similar to the FAST FTL [Lee et al. 2007]. We integrate silent eviction with background and foreground garbage collection for data blocks, and with merge operations for SSC-V when recycling log blocks [Aayush et al. 2009]. 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.

5.2. SSC Prototype Implementation

We use the OpenSSD platform [OpenSSD Project Website 2013] (see Figure 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 the SSC design to verify its practicality and validate if it performs as we projected in simulation.

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 [OpenSSD Project Website 2013]. The board is composed of commodity SSD parts (see Table III): 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 mapping and for SATA buffers; and 8 slots holding up to 256GB 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 two planes on two 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 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

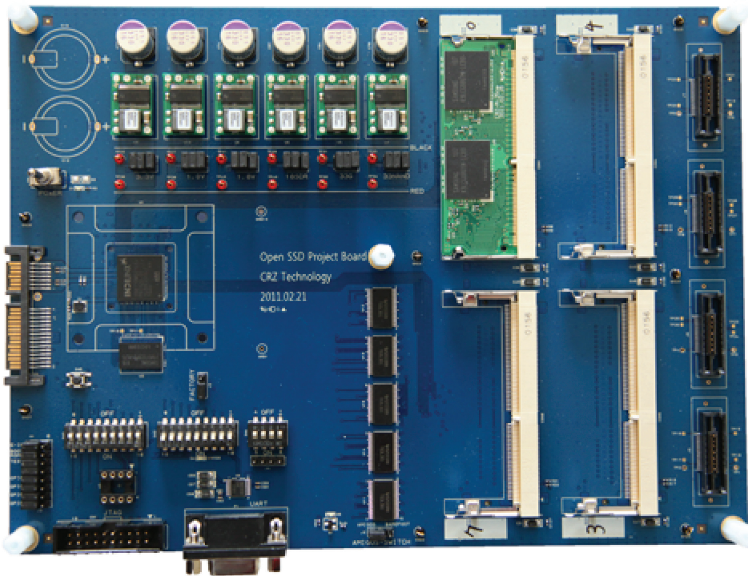

Fig. 3. OpenSSD Architecture. Major components of OpenSSD platform are the Indilinx Barefoot SSD controller, internal SRAM, SDRAM, NAND flash, specialized hardware for buffer management, flash control, and memory utility functions; and debugging UART/JTAG ports.

Table III. OpenSSD Device Configuration

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

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: 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 logic 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 because it provides support for garbage collection.

5.3. 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 Section 6. Here, we list

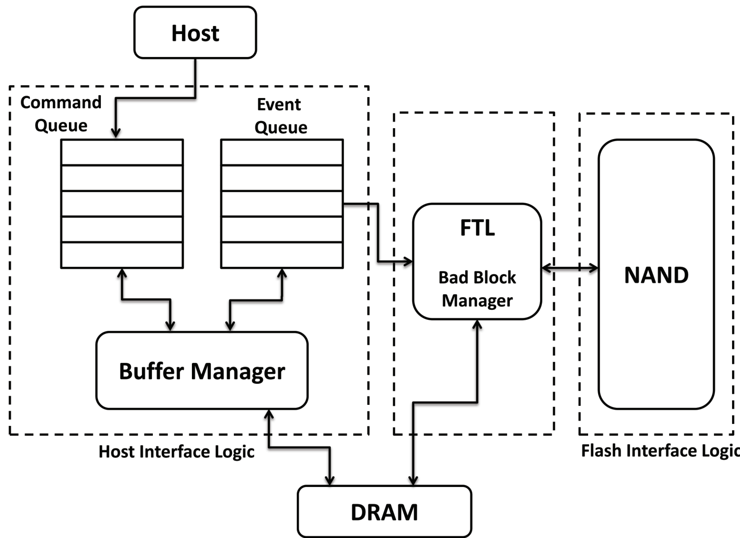

Fig. 4. OpenSSD Internals. Major components of OpenSSD internal design are host interface logic, flash interface logic and flash translation layer.

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 prototype 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 for SSC. Page-map FTLs are more fine-grained and faster than hybrid FTLs, which are commonly used in commodity SSDs. As a result, we prototype the *SSC-P* (solid-state cache with *p*age-mapped FTL) variant of the silent eviction policy.
—We use a linear hash-map to store the address translation map in device memory on OpenSSD because it lacks memory allocation support in firmware for implementing the sparse hash-map data structure.

6. SSC PROTOTYPE: EXPERIENCES & LESSONS

FlashTier's Solid-State Cache (SSC) design [Saxena et al. 2012] improves caching performance with changes to the block interface, flash management algorithms within the device, and the OS storage stack. Therefore, the SSC design is an ideal candidate for prototyping and studying the disruptive nature of such systems.

In this section, 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. This is the first documentation in existing research and commercial literature on: implementing new flash interfaces in OS storage stack and device firmware, and internal details of commodity SSD hardware designs to manage raw flash.

First, we found that passing new commands from the file-system layer through the Linux storage stack and into the device firmware raised substantial engineering

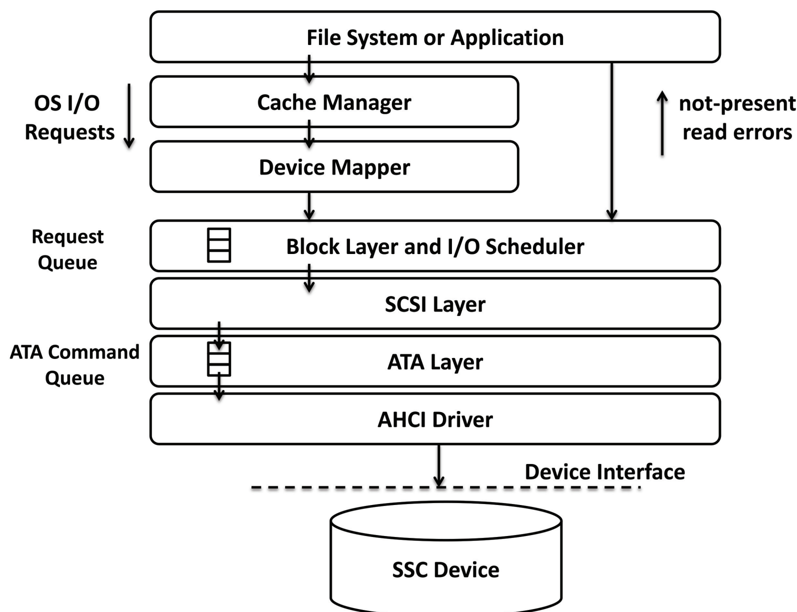

Fig. 5. OS Interfaces. I/O path from the cache manager or file system to the device through different storage stack layers.

hurdles. For example, the I/O scheduler must know which commands can be merged and reordered. In Section 6.1, 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.

Second, 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 6.2 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.

6.1. OS and Device Interfaces

A key challenge in implementing the SSC is that its design changes the *interface* to the device by adding new commands and new 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.

6.1.1. 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.

First, the forward commands from the OS to the device pass through several layers in the OS, shown in Figure 5, 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 [Gunawi et al. 2007; Prabhakaran et al. 2005].

Third, the OpenSSD platform provides hardware support for the SATA protocol (see Figure 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.

6.1.2. 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 subtypes of existing commands. We modified the I/O scheduler to only merge requests with the same command and subtype. For the SSC, a *write-dirty* command is not merged with a *write-clean* operation. The SCSI or ATA layers blindly pass the subtype 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 subtype 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.

6.1.3. 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.

6.2. 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 such as the SSC consistency mechanisms, unified address space and silent cache eviction was far more difficult.

We now describe the problems and solutions associated with implementing these mechanisms.

Consistency and Durability. The 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.

The FlashTier design uses checkpoints to reduce the time to reconstruct a page map on startup (refer Section 4.2). 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 prototype 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 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. Our design supports sparse address spaces with a memory-optimized data structure based on perfect hash function [Google, Inc. 2012]. 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 prototype 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. SSC uses 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, SSC 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. We reserve 7% of the device's capacity to accommodate bad blocks, metadata (e.g., for logging and checkpointing as described previously), 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.

6.2.1. 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 to 20 MB to leverage the internal way and channel

Table IV. Simulation Parameters

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

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.

7. EVALUATION

We compare the cost and benefits of FlashTier's design components against traditional caching on SSDs and focus on four 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?
—What are the benefits of SSC design for arbitrary workloads on a real hardware prototype?

We describe our methods for simulation and prototype evaluation, and present a summary of our results before answering these questions in detail.

7.1. Methodology

Simulation Methodology. We simulate SSD, SSC and SSC-V with the parameters in Table IV, which are taken from the latencies of the third generation Intel 300 series SSD [Intel Corp. 2012]. 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-V device 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 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 [OCZ Technologies 2012].

We compare the FlashTier system against the *Native* system, which 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 block traces with the characteristics shown in Table V. 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) [Koller and Rangaswami 2010]; and a small enterprise data center hosting user home directories (*usr* workload) and project directories (*proj* workload) [Narayanan et al. 2008]. 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

Table V. Workload Characteristics

Workload	Range	Unique Blocks	Total 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

All requests are sector-aligned and 4,096 bytes.

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 [Facebook, Inc. 2013]. 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 has been recently included in Facebook's FlashCache software, upon which we based our cache manager. We evaluate using standard workload profiles with the filebench benchmark. Unlike trace replay used for simulation, the filebench workloads allow us to accurately model inter-request ordering and timing dependencies. In addition to read- and write-intensive workloads as used in simulation, we also use filebench workloads with fsync operations to evaluate the SSC prototype.

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 OpenSSD. 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.

7.2. 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 6 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 7.6, which is offset by the cost of FlashTier's consistency guarantees, which are described in Section 7.5.

Memory Consumption. Table VI 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 7.4.

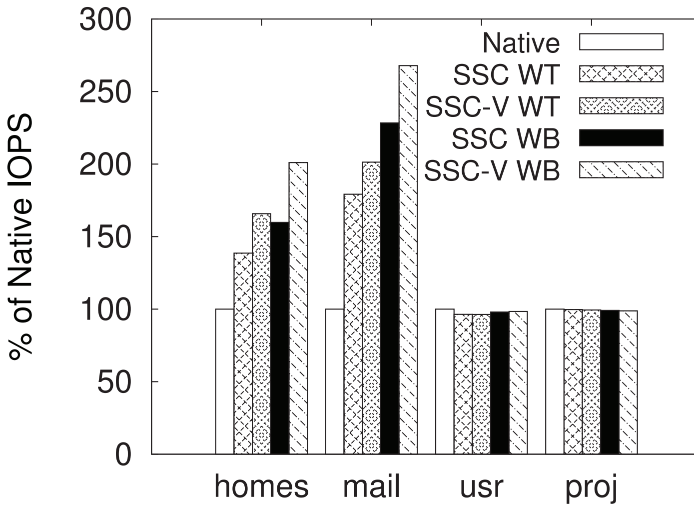

Fig. 6. Simulated Performance. The performance of write-through and write-back FlashTier systems normalized to native write-back performance. We do not include native write-through because it does not implement durability.

Table VI. Memory Consumption

	Size	SSD	SSC	SSC-V	Native	FTCM
Workload	GB	Device (MB)			Host (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

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.

Wearout. For Figure 6, 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 7.6.

7.3. Prototype System Comparison

Performance. Figure 7 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. 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

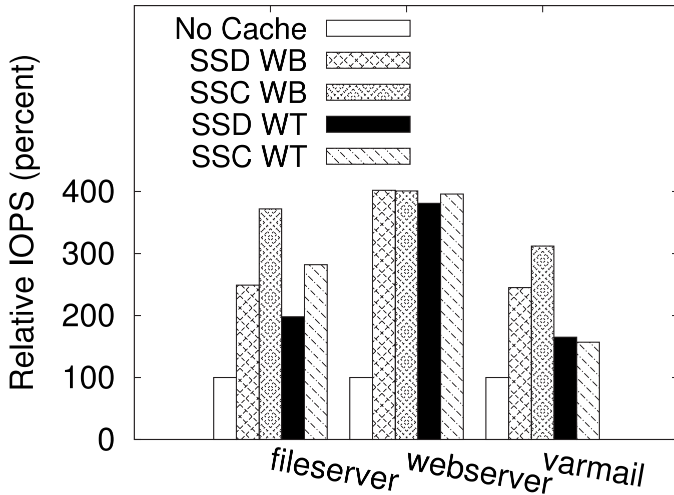

Fig. 7. SSC Prototype Performance. The performance of write-back and write-through caches using SSD and SSC relative to no-cache disk execution.

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 or silent eviction benefit, which repeats FlashTier's simulated 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 preceeding ones reported (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 used in high-end SSDs, instead of the simulated hybrid FTL used in commodity SSDs, so garbage collection is less expensive. We do not use the simulated traces because they can not be replayed with correct ordering and timing dependencies on the SSC prototype.

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. Our results with OpenSSD correlate with the simulated results. The major difference in functionality between the two systems is different consistency 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%.

7.4. 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 VI 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, 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 VI 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

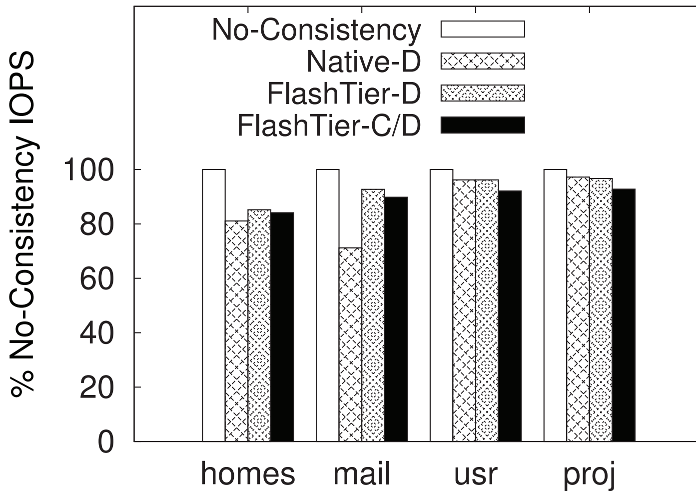

Fig. 8. Consistency Cost. No-consistency system does not provide any consistency guarantees for cached data or metadata. Native-D and FlashTier-D systems only provide consistency for dirty data. FlashTier-C/D provides consistency for both clean and dirty data.

(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 [Fusion-io Inc. 2013c], these savings are immediately realizable.

7.5. 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 nonvolatile 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 the OpenSSD prototype in Section 7.3. 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 8 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 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 [Aayush et al. 2009].

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% to 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% to 15% for FlashTier-D and 11% to 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% to 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% to 37% because of frequent small metadata writes. Both FlashTier configurations increase response time less, by 18% to 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% to 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 9 compares the time for recovering after a crash. The mapping and cache sizes for each workload are shown in Table VI.

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 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 [Aayush et al. 2009], 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.

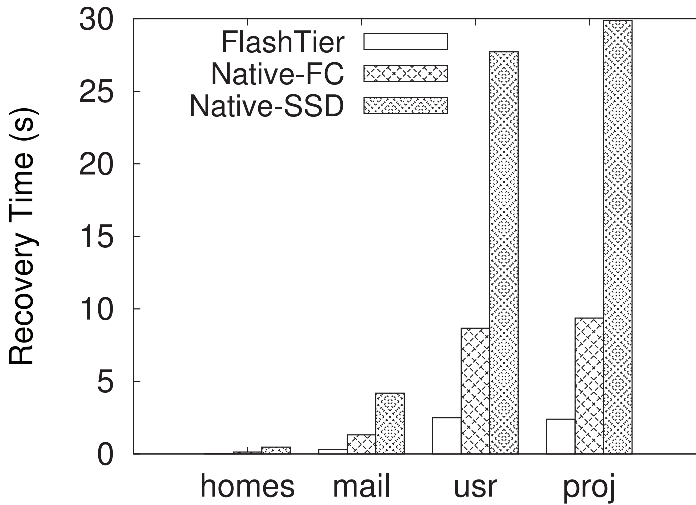

Fig. 9. Recovery Time. Native-FC accounts for only recovering FlashCache cache manager state. Native-SSD accounts for only recovering the SSD mapping.

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*.

7.6. 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 VI. 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 7.3. Those end-to-end results validate our findings for simulation in this section.

Garbage Collection. Figure 10 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 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 VII. 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, having more log blocks

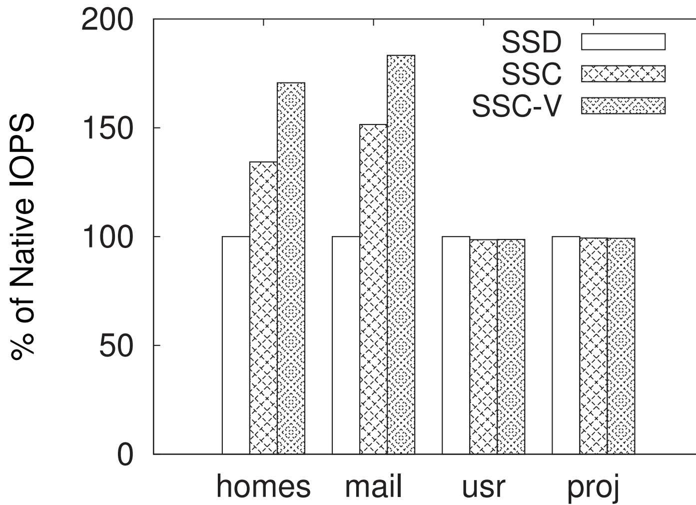

Fig. 10. Garbage Collection Performance. Comparing the impact of garbage collection on caching performance for different workloads on SSD, SSC and SSC-V devices.

Table VII. Wear Distribution

	Erases			Wear Diff.			Write Amp.			Miss Rate		
Workload	SSD	SSC	SSC-V	SSD	SSC	SSC-V	SSD	SSC	SSC-V	SSD	SSC	SSC-V
homes	878,395	829,356	617,298	3,094	864	431	2.30	1.84	1.30	10.4	12.8	11.9
mail	880,710	637,089	525,954	1,044	757	181	1.96	1.08	0.77	15.6	16.9	16.5
usr	339,198	369,842	325,272	219	237	122	1.23	1.30	1.18	10.6	10.9	10.8
proj	164,807	166,712	164,527	41	226	17	1.03	1.04	1.02	9.77	9.82	9.80

For each workload, the total number of erase operations, the maximum wear difference between blocks, the write amplification, and the cache miss rate is shown for SSD, SSC and SSC-V.

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 VII, 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 previously 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 VII 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 drops 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.

8. RELATED WORK

The FlashTier design draws on past work investigating the use of solid-state memory for caching and hybrid systems, new flash interfaces and hardware prototypes.

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 [Archer 2006; Gregg 2008]. Oracle has a write-through flash cache for databases [Oracle Corp. 2012] 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 [Facebook, Inc. 2013; Mack 2012]. Storage vendors have also proposed the use of local SSDs as write-through caches to centrally-managed storage shared across virtual machine servers [Byan et al. 2011; NetApp, Inc. 2013; Zhang et al. 2013; Koller et al. 2013]. 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.

New Flash 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 [Zhang et al. 2012], providing caching support [Nellans et al. 2011; Fusion-io, Inc. 2013a; Intel

Corp. 2011; Mesnier et al. 2011], exposing key-value interfaces [Fusion-io, inc. 2013b; Balakrishnan et al. 2013], and completely delegating block allocation to the SSD [Josephson et al. 2010]. 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 commercial read/write/trim space without requiring migration callbacks from the device into the OS like these systems.

Hybrid Systems. SSD vendors have recently proposed new flash caching products, which cache most-frequently accessed reads and write I/O requests to disk [Fusion-io, Inc. 2013a; OCZ 2012]. FlashCache [Kgil and Mudge 2006] and the flash-based disk cache [Roberts et al. 2009] also propose specialized hardware for caching. Hybrid drives [Bisson 2007; Apple, Inc. 2013] 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 [Yadgar et al. 2007; Wong and Wilkes 2002]. 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 [Mesnier et al. 2011; Guerra et al. 2011]. 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.

Hardware Prototypes. Research platforms for characterizing flash performance and reliability have been developed in the past [Boboila and Desnoyers 2010; Bunker et al. 2012; Davis and Zhang 2009; Lee et al. 2010, 2009]. In addition, there have been efforts on prototyping phase-change memory based prototypes [Akel et al. 2011; Caulfield et al. 2010]. 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 [OpenSSD Project Participants 2013; VLDB Lab 2012] and educational purposes [Computer Systems Laboratory, SKKU 2012]. Furthermore, we investigated changes to the flash-device interface, while past work looks at internal FTL mechanisms.

9. CONCLUSIONS

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 article, 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.

We implemented the SSC design and interfaces on a hardware prototype, which was a substantial effort, yet ultimately proved its value by providing a concrete demonstration of the performance benefits of FlashTier's SSC design. In addition, we document the first experiences with managing raw flash and internals of commercial SSD hardware such as large page sizes, flash channel and bank parallelism, and hardware acceleration of I/O commands.

In the near future, 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.

REFERENCES

Gupta Aayush, Kim Youngjae, and Urgaonkar Bhuvan. 2009. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of ASPLOS*.

Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John Davis, Mark Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of USENIX ATC*.

Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. 2011. Onyx: A protoype phase-change memory storage array. In *Proceedings of HotStorage*.

Apple Inc. 2013. Fusion drive. http://www.apple.com/imac/performance/#fusion.

Tom Archer. 2006. MSDN Blog: Microsoft ReadyBoost. http://blogs.msdn.com/tomarcher/archive/2006/06/02/615199.aspx.

Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Micheal Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed data structures over a shared log. In *Proceedings of SOSP*.

Timothy Bisson. 2007. Reducing hybrid disk write latency with flash-backed IO requests. In *Proceedings of MASCOTS*.

Simona Boboila and Peter Desnoyers. 2010. Write endurance in flash drives: Measurements and analysis. In *Proceedings of FAST*.

Trevor Bunker, Michael Wei, and Steven Swanson. 2012. Ming II: A flexible platform for NAND flash-based research. Technical Report, TR CS2012-0978. University of California, San Diego.

Steve Byan, James Lentini, Luis Pabon, Christopher Small, and Mark W. Storer. 2011. Mercury: Host-side flash caching for the datacenter. In *Proceedings of FAST (Poster)*.

Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of IEEE Micro*.

Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A Content-Aware Flash Translation Layer enhancing the lifespan of flash memory-based solid state drives. In *Proceedings of FAST*.

Computer Systems Laboratory, SKKU. 2012. Embedded systems design class. http://csl.skku.edu/ICE3028S12/Overview.

Jonathan Corbet. 2008. Barriers and journaling filesystems. http://lwn.net/Articles/283161/.

John D. Davis and Lintao Zhang. 2009. FRP: A nonvolatile memory research platform targeting NAND flash. In *Proceedings of the Workshop on Integrating Solid-State Memory into the Storage Hierarchy, ASPLOS*.

Scott Doyle and Ashok Narayan. 2010. Enterprise solid state drive endurance. In *Proceedings of Intel IDF*.

EMC. 2013. Fully Automated Storage Tiering (FAST) Cache. http://www.emc.com/corporate/glossary/fully-automated-storage-tiering-cache.htm.

Facebook, Inc. 2013. Facebook FlashCache. https://github.com/facebook/flashcache.

Fusion-io, Inc. 2013a. directCache. http://www.fusionio.com/data-sheets/directcache.

Fusion-io, Inc. 2013b. ioMemory Application SDK. http://www.fusionio.com/products/iomemorysdk.

Fusion-io, Inc. 2013c. ioXtreme PCI-e SSD Datasheet. (2013). http://www.fusionio.com/ioxtreme/PDFs/ioXtremeDS_v.9.pdf.

Google, Inc. 2012. Google sparse hash. http://goog-sparsehash.sourceforge.net.

Brendan Gregg. 2008. Sun Blog: Solaris L2ARC Cache. http://blogs.oracle.com/brendan/entry/test.

Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. 2011. Cost effective storage using extent based dynamic tiering. In *Proceedings of FAST*.

Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2007. Improving file system reliability with I/O shepherding. In *Proceedings of SOSP*. 293–306.

Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. 2011. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of FAST*.

Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, and Phil Garrett. 1988. Group commit timers and high-volume transaction systems. Tandem Technical Report 88.1.

Intel. 1998. Understanding the Flash Translation Layer (FTL) specification. Application Note AP-684.

Intel Corp. 2011. Intel smart response technology. http://download.intel.com/design/flash/nand/325554.pdf.

Intel Corp. 2012. Intel 300 series SSD. http://ark.intel.com/products/family/56542/Intel-SSD-300-Family.

William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. 2010. DFS: A file system for virtualized flash storage. In *Proceedings of FAST*.

T. Kgil and Trevor N. Mudge. 2006. FlashCache: A NAND flash memory file cache for low power web servers. In *Proceedings of CASES*.

Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. 2009. FlashSim: A simulator for NAND flash-based solid-state drives. In *Proceedings of the International Conference on Advances in System Simulation,* 125–131.

Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. 2013. Write policies for host-side flash caches. In *Proceedings of FAST*.

Ricardo Koller and Raju Rangaswami. 2010. I/O Deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of FAST*.

Sungjin Lee, Kermin Fleming, Jihoon Park, Keonsoo Ha, Adrian M. Caulfield, Steven Swanson, Arvind, and Jihong Kim. 2010. BlueSSD: An open platform for cross-layer experiments for NAND flash-based SSDs. In *Proceedings of the Workshop on Architectural Research Prototyping*.

Sungjin Lee, Keonsoo Ha, Kangwon Zhang, Jihong Kim, and Junghwan Kim. 2009. FlexFS: A flexible flash file system for MLC NAND flash memory. In *Proceedings of Usenix ATC*.

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

Michael Mesnier, Jason B. Akers, Feng Chen, and Tian Luo. 2011. Differentiated Storage Services. In *Proceedings of SOSP*.

Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. In *Proceedings of FAST*.

David Nellans, Michael Zappe, Jens Axboe, and David Flynn. 2011. ptrim() + exists(): Exposing new FTL primitives to applications. In *Proceedings of NVMW*.

NetApp, Inc. 2013. Flash cache for enterprise. http://www.netapp.com/us/products/storage-systems/flash-cache.

OCZ. 2012. OCZ synapse cache SSD. http://www.ocztechnology.com/ocz-synapse-cache-sata-iii-2-5-ssd.html.

OCZ Technologies. 2012. Vertex 3 SSD. http://www.ocztechnology.com/ocz-vertex-3-sata-iii-2-5-ssd.html.

Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2012. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *Proceedings of FAST*.

OpenSSD Project Participants. 2013. Participating Institutes. http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform.

OpenSSD Project Website. 2013. Indilinx Jasmine platform. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

Oracle Corp. 2012. Oracle database smart flash cache. http://www.oracle.com/technetwork/articles/systems-hardware-architectur%e/oracle-db-smart-flash-cache-175588.pdf.

Xiangyong Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. 2011. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of HPCA*. 301–311.

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of SOSP*. 206–220.

Vijayan Prabhakaran, Thomas Rodeheffer, and Lidong Zhou. 2008. Transactional flash. In *Proceedings of OSDI*.

David Roberts, Taeho Kgil, and Trevor Mudge. 2009. Integrating NAND flash devices onto servers. *Commun. ACM* 52, 4, 98–106.

Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1.

Ryan Mack. 2012. Building Facebook timeline: Scaling up to hold your life story. https://www.facebook.com/note.php?note_id=10150468255628920.

Mohit Saxena and Michael M. Swift. 2010. FlashVM: Virtual memory management on flash. In *Proceedings of Usenix ATC*.

Mohit Saxena, Michael M. Swift, and Yiying Zhang. 2012. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of EuroSys*.

Mohit Saxena, Yiying Zhang, Michael M. Swift, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Getting real: Lessons in transitioning research simulations into hardware systems. In *Proceedings of FAST*.

STEC, Inc. 2012. EnhanceIO. https://github.com/stec-inc/EnhanceIO.

VLDB Lab. 2012. SKKU University, Korea. http://ldb.skku.ac.kr.

Theodore M. Wong and John Wilkes. 2002. My cache or yours? Making storage more exclusive. In *Proceedings of Usenix ATC*.

Michael Wu and Willy Zwaenepoel. 1994. eNVy: A non-volatile, main memory storage system. In *Proceedings of ASPLOS-VI*.

Gala Yadgar, Michael Factor, and Assaf Schuster. 2007. Karma: Know-it-all replacement for a multilevel cache. In *Proceedings of FAST*.

Yiying Zhang, Leo Prasath Arulraj, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2012. De-indirection for Flash-based SSDs with nameless writes. In *Proceedings of FAST*.

Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Warming up storage-level caches with bonre. In *Proceedings of FAST*.

Received July 2013; accepted January 2014