Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems

Mohit Saxena, Yiying Zhang
Michael M. Swift, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison
{msaxena,yyzhang,swift,dusseau,remzi}@cs.wisc.edu

Abstract

Flash-based solid-state drives have revolutionized storage with their high performance. Their sophisticated internal mechanisms have led to a plethora of research on how to optimize applications, file systems, and internal SSD designs. Due to the closed nature of commercial devices though, most research on the internals of an SSD, such as enhanced flash-translation layers, is performed using simulation or emulation. Without implementation in real devices, it can be difficult to judge the true benefit of the proposed designs.

In this paper, we describe our efforts to implement two new SSD designs that change both the internal workings of the device and its interface to the host operating system. Using the OpenSSD Jasmine board, we develop a prototype of FlashTier's Solid State Cache (SSC) and of the Nameless Write SSD. While the flash-translation layer changes were straightforward, we discovered unexpected complexities in implementing extensions to the storage interface.

We describe our implementation process and extract a set of lessons applicable to other SSD prototypes. With our prototype we validate the performance claims of FlashTier and show a 45-52% performance improvement over caching with an SSD and a 90% reduction in erases.

1 Introduction

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 [24, 27]), optimizing their internal algorithms (*e.g.*, garbage collection [7, 16, 17]), and extending their interface (*e.g.*, caching operations [32]). However, most research looking at internal SSD mechanisms relies on simulation rather than direct experimentation [2, 27, 32, 36].

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 [18, 26, 11, 12], who release little about the internal workings of their devices. This limits the opportunities for research innovation on new flash interfaces, on OS designs to better integrate flash in the storage hierarchy, and on adapting the SSD internal block management for application requirements.

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

We sought to validate two recently proposed SSD designs by implementing them as hardware prototypes. FlashTier's Solid-State Cache (SSC) design [32] improves caching performance with changes to the block interface, flash management algorithms within the device, and the OS storage stack. Nameless Writes [36] simplifies garbage collection and improves write performance by moving the address-mapping function out of the device and into the file system, and requires changes to the same components as SSCs plus the file system. Therefore, these designs are ideal candidates for studying the disruptive nature of such systems.

We prototype both systems with the OpenSSD Jasmine hardware platform [33]. 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 FTL algorithms. As a real storage device with performance comparable to commercial SSDs, it allows us to test new SSD designs with existing file-system benchmarks and real application workloads.

During prototyping, we faced several challenges not foreseen in published work on new flash interfaces. First,

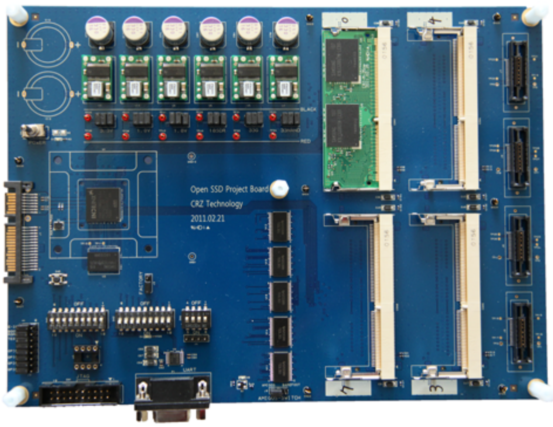

Figure 1: **OpenSSD Architecture:** Major components of OpenSSD platform are the Indilinx Barefoot SSD controller, internal SRAM, SDRAM, NAND flash, specialized hardware for buffer management, flash control, and memory utility functions; and debugging UART/JTAG ports.

we found that the baseline performance of the OpenSSD board was low, and that the effect of new designs was drowned out by other performance problems. We introduced a *merge buffer* to align writes to internal boundaries and a *read buffer* for efficient random reads.

Second, we found that passing new commands from the file-system layer through the Linux storage stack and into the device firmware raised substantial engineering hurdles. For example, the I/O scheduler must know which commands can be merged and reordered. We developed novel techniques to tunnel new commands through the storage stack and hardware as variations of existing commands, which limits software changes to the layers at the ends of the tunnel. For example, we return cache-miss errors in the data field of a read.

Third, we encountered practical hardware limitations within the firmware. The OpenSSD board lacks an accessible out-of-band (OOB) area, and has limited SRAM and DRAM within the device. This made us redesign 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.

For the SSC design, we implemented the interface functionality entirely within the firmware running on the OpenSSD board. For Nameless Writes, the challenges in passing data between the device and the host OS led us to a split-FTL design. A minimal FTL on the device exports primitive operations, while an FTL within the host OS uses these primitives to implement higher-level functionality. This split design simplifies the FTL implementation and provides a convenient mechanism to work around hardware limitations, such as limited DRAM or fixed-function hardware.

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

Table 1: **OpenSSD device configuration.**

With our SSC prototype, we validate that FlashTier's projected benefits, including better write performance and reliability, are indeed possible. Compared to caching with an SSD, the SSC design provides 45–52% better write performance. Our Nameless-Write prototype, demonstrates that the split-FTL approach may be a useful method of implementing new interface designs relying on upcalls from an SSD to host.

2 Background

We use the OpenSSD platform [33] as 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 our own designs — FlashTier SSC and Nameless Writes SSD — to verify their practicality and validate if they perform as we projected in simulation earlier.

2.1 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 [33]. The board is composed of commodity SSD parts: an Indilinx Barefoot ARM-based SATA controller, introduced in 2009 for second generation SSDs and still used in many commercial SSDs; 96 KB SRAM; 64 MB DRAM for storing the flash translation mapping and for SATA buffers; and 8 slots holding up to 256 GB of MLC NAND flash. The controller runs firmware that can send read/write/erase and copyback (copy data within a bank) operations to the flash banks over a 16-bit I/O channel. The chips use two planes and have 8 KB physical pages. The device uses large 32 KB virtual pages, which improve performance by striping data across physical pages on 2 planes on 2 chips within a flash bank. Erase blocks are 4 MB and composed of 128 contiguous virtual pages.

The controller provides hardware support to accelerate command processing in the form of command queues and a buffer manager. The command queues provide a FIFO for incoming requests to decouple FTL operations from receiving SATA requests. The hardware provides separate read and write command queues, into which arriving commands can be placed. The queue provides a *fast path* for performance-sensitive commands. Less common commands, such as *ATA flush*, *idle* and *standby* are executed on a *slow path* that waits for all queued

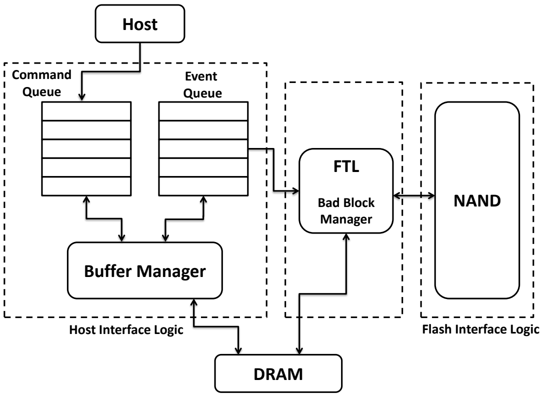

Figure 2: **OpenSSD Internals:** Major components of OpenSSD internal design are host interface logic, flash interface logic and flash translation layer.

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

2.2 FlashTier: Solid-State Cache (SSC)

The FlashTier SSC is a new interface to flash storage purpose-built for caching [32]. The internal flash management algorithms and device interface of SSC offload caching functionality from the host operating system with cache-specific commands. The SSC interface distinguishes clean data, which can be deleted from the device silently, from dirty data, which cannot. Its interface consists of six commands: *write-clean* and *write-dirty* to write clean and dirty data, respectively; *read* to read data or return an error if it is not present; *evict* to remove data from the cache, *clean* to convert dirty data to clean after it has been written back to a disk, and *exists* to test whether blocks are currently dirty. The SSC interface provides consistency and durability guarantees within the device to provide a warm cache after a system reboot or crash. An SSC provides a *unified address space* for directly translating disk addresses to physical flash addresses, rather than requiring host software to map disk addresses to logical flash address that the FTL translates to physical flash addresses. In addition, SSCs provide *silent eviction* with which they can evict clean data without notifying the host as an alternative to garbage collection.

FlashTier relies on a block-level cache manager to interpose on disk requests from the file system and transparently shunt them to the SSC. If the SSC does not contain the data, the manager sends the request to the backing disk. Compared to caching with an SSD, FlashTier promises better write performance, because it can evict data silently instead of performing expensive garbage collection; reduced memory on the host, because the disk-block-to-flash-location mapping is stored in the SSC; and better reliability, again by avoiding expensive copies for garbage collection.

2.3 Nameless Writes

Nameless Writes [36] are a new device interface that removes the internal mapping function of an FTL and instead exposes physical flash addresses to the file system. A Nameless-Write SSD chooses the location to write data and sends back the physical address to the file system, which stores it in its metadata. This allows the device to control block allocation, enabling it to execute critical tasks such as garbage collection and wear leveling, but removes the need for large and costly mapping tables. Nameless Writes retain the standard reads, write and trim commands, which it uses for file-system metadata, and introduces three new commands. Two commands operate on physical addresses: *nameless-write* writes data at an unspecified address and returns the new physical address, and *physical-read* reads data by physical address. Finally, the device can send a *migration-upcall* request to the host indicating that it has relocated data as part of wear leveling or garbage collection so the file system can update its metadata. Nameless Writes change the device FTL, SSD interface, and file system to move logic for allocating data blocks out of the file system and into the device.

3 Implementation Experiences

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

3.1 Baseline Performance

The biggest challenge with prototyping new SSD designs was to first improve the performance of the baseline platform to its maximum. This enables us to run real workloads and get realistic results comparable to commercial SSDs. The OpenSSD hardware is capable

Workload	Baseline	OS Req.		IO/s	
		Size			
Request Size	IO/s	(sectors)	Cause	Intel 520	Disk
4 KB	2,560	9	Partial	22,528	1,497
8 KB	2,048	18	Partial	17,280	826
16 KB	1,920	36	Partial	11,008	563
32 KB	2,912	82	Align	7,136	419
64 KB	1,456	150	Par.	3,824	288
128 KB	648	256	Par.	2,056	212
256 KB	248	257	Cont.	988	149
512 KB	132	266	Cont.	500	98
Seq 4 KB	15,616	336	Partial	61,950	24,437
Seq 4+32 KB	20,736	64	Align	62,120	22,560

Table 2: **Write Performance**: Random write performance (top) and sequential write performance (bottom) as compared to an Intel 520 SSD and a disk. Columns 3 shows the average number of 512 B sectors per request sent to the device, and Column 4 summarizes the cause of poor performance (*Partial* = partial writes, *Align* = unaligned writes, *Par* = insufficient parallelism, *Cont* = bank contention).

of 90 MB/sec writes and 125 MB/sec reads. However, the reference FTL implementations delivered this performance only with specific workloads, as we describe below. For common cases such as 4 KB random writes, performance dropped to 10 MB/sec. Thus, any performance benefit we achieved through FlashTier or Nameless Writes would be overwhelmed by poor baseline performance. The first task of prototyping a new interface to flash, ironically, was to make the existing interface perform well. We note that many of the challenges we faced are known, and here we describe the effort required to implement solutions.

3.1.1 Problems

We began by analyzing the peak performance achievable by the hardware and comparing the application performance delivered by the FTL for different workload parameters. We perform tests on an 8 GB partition on an empty device. For read tests, data was written sequentially. Table 2 shows I/O Operations/sec (IOPS) with varying random and sequential request sizes in column 2. All measurements were made with the *fio* microbenckmarking tool [22]. For comparison, the last two columns show performance of the Intel 520 120 GB SSD, and for a Western Digital WD1502FAEX 7200 RPM 1.5 TB disk. Overall, random IOPS peaks at 32 KB, the virtual page size of the device, and degrades with larger requests. Smaller requests also suffer reduced IOPS and much lower bandwidth. Read performance, shown in Figure 4 was 125 MB/sec for sequential reads but only 2,340 IOPS for random reads.

Alignment and Granularity. The smallest unit of I/O within the FTL is a 32 KB virtual page. For smaller requests, the firmware has to read the old data, merge in the new data, and finally write the updated virtual page. Therefore, writes smaller than 32 KB suffer write amplification from this read-modify-write cycle. For example, 8 KB blocks achieve only 2,048 IOPS (16 MB/s), while 32 KB blocks achieve 2,912 (91 MB/s). Furthermore, writes that are not aligned on 32 KB boundaries can suffer a read-modify-write cycle for *two* blocks.

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

OS Merging. A final source of problems is the I/O scheduler in Linux. As noted above, large sequential requests degrade performance. Linux I/O schedulers — NOOP, CFQ and Deadline — merge spatially adjacent requests. As shown in column 3 (Req Size) of Table 2, for the Seq 4 KB row the OS merges an average of 336 sectors (168 KB) for sequential 4 KB writes and achieves only 61 MB/sec. When we artificially limit merging to 32 KB (64 sectors), shown in the last row (labeled Seq 4 KB+32), performance improves to 81 MB/sec. This is still below peak rate because the requests may not all be aligned to virtual page boundaries.

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

3.1.2 Solutions

Write performance. We address poor write performance by implementing a *merge buffer* within the FTL to stage data before writing it out. This ensures that data can be written at virtual-page granularity rather than at the granularity supplied by the operating system. Data for write commands are enqueued in a FIFO order in the SATA write buffer, from which the FTL copies it to a merge buffer. The FTL flushes the buffer only when the buffer is full to ensure there are no partial writes. The buffer is striped across different flash banks to minimize the idle time when a request waits for a free bank. This also improves sequential write performance by nullifying the impact of partial or unaligned writes for requests merged by the I/O scheduler.

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

Figure 3 shows the write performance of the OpenSSD board using the reference firmware and page-mapped FTL implementation (baseline), and with merging and

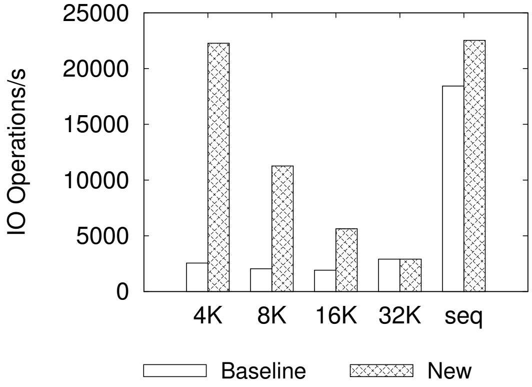

Figure 3: **Small Requests**: Impact of merge buffering on write performance.

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

Read Performance. Sequential read performance of the baseline FTL is excellent (125 MB/sec) because virtual pages contained contiguous data. However, random reads perform similar to random writes and achieve only 2,340 IOPS. Using a merge buffer can hurt sequential reads if data is written randomly, because a 32 KB virtual page may contain a random selection of data.

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

Figure 4 shows the overall performance of the baseline and new systems for 4 KB requests. For writes, the new system has improved performance for both random and sequential access patterns as shown in Figure 3. For sequential reads, the new system is 30% slower because of the delay added by the read buffer. For random 4 KB requests, the new system is twice as fast as the baseline, largely because of the optimization to reply as soon as only the requested data has been copied. The baseline system, in contrast, waits for an entire 32 KB virtual page on every 4 KB read. Thus, we sacrifice some sequential performance to improve random performance, which is likely to be more beneficial for a cache.

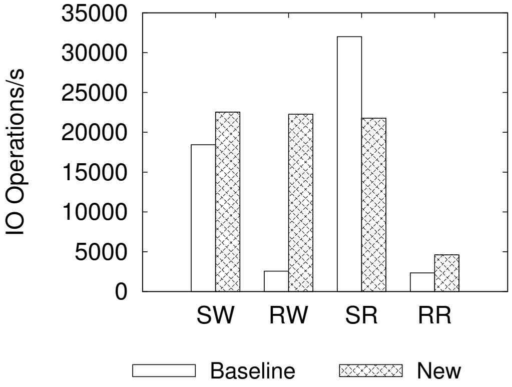

Figure 4: **Read/Write Performance:** On baseline unoptimized and new optimized OpenSSD FTLs for fio benchmark with 4 KB request sizes.

3.1.3 Lessons Learned

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

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

3.2 OS and Device Interfaces

A key challenge in implementing Nameless Writes and the SSC is that both designs change the *interface* to the device by adding new commands, new command responses, and in the case of Nameless Writes, unrequested up-calls. 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.

3.2.1 Problems

We identified three major problems while implementing support for SSC and Nameless SSD interfaces in the OS and device: how to get new commands through the OS

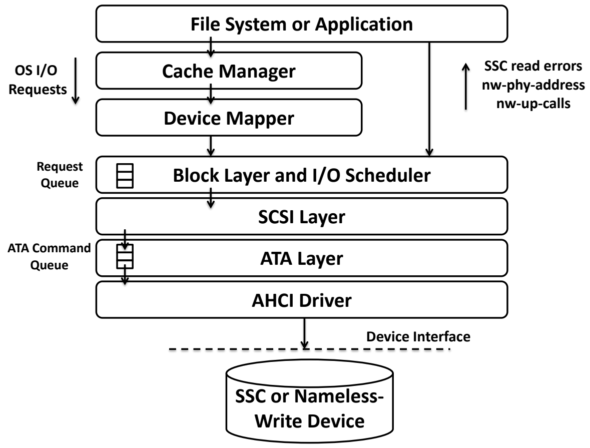

Figure 5: **OS Interfaces:** I/O path from the cache manager or file system to the device through different storage stack layers.

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, and the nameless interfaces returns the physical address for data following a nameless write. 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 [15, 28]. In addition, the SATA write command response has no fields with which an address can be returned, so returning the physical location of data is difficult.

Third, the OpenSSD platform provides hardware support for the SATA protocol (see Figure 2) 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.

3.2.2 Solutions

We developed several general techniques for introducing new commands. We defer discussion of the detailed mechanisms for Nameless Writes and SSCs to the following sections.

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

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

OpenSSD request handling. Within the device, commands arrive from the SATA bus and are then enqueued by the host-interface firmware. The FTL asynchronously pulls requests from the queues to be processed. Thus, the key change needed for new requests is to communicate the command type from arriving commands to the FTL, which executes commands. We borrow two bits from the length field of the request (a 32-bit value) to encode the command type. The FTL decodes these length bits to determine which command to execute, and invokes the function for the command. This encoding ensures that the OpenSSD hardware uses the fast path for new variations of read and writes, and allows multiple variations of the commands.

Reverse-path device responses. The key challenge for the SSC implementation is to indicate that a read request missed in the cache without returning an error, which causes the AHCI and ATA layers to retry the request or shutdown the device. Thus, we chose to return a read miss in the data portion of a read as a distinguished pattern; the FTL copies the pattern into a buffer that is returned to the host rather than reading data from flash. The cache manager system software accessing the cache can then check whether the requested block matches the read-miss pattern, and if so consult the backing disk.

This approach is not ideal, as it could cause an unnecessary cache miss if a block using the read-miss pattern is ever stored. In addition, it raises the cost of misses, as useless data must be transferred from the device.

We defer the discussion of the reverse path for Nameless Writes to Section 3.4, as they demand even greater changes to the storage stack.

3.2.3 Lessons Learned

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

- The OS storage stack's layered design may require each layer to act differently for the introduction of a new forward command. For example, new commands must have well-defined semantics for request schedulers, such as which commands can be combined and how they can be reordered.
- SSDs hardware and firmware must evolve to support new interfaces. For example, the hardware command queue and the DMA controller are built for standard interfaces and protocols, which requires us to mask the command type in the firmware itself to still use hardware accelerated servicing of commands.
- The device response paths in the OS are difficult to change, so designs that radically extend existing communication from the device should consider data that will be communicated. For example, most storage code assumes that errors are rare and catastrophic. Interfaces with more frequent errors, such as a cache that can miss, must address how to return benign failures. It may be worthwhile to investigate overloading such device responses on data path for SATA/SAS devices.

3.3 SSC Implementation

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

3.3.1 Interfaces

The data access routines, *read*, *write-dirty* and *write-clean* are similar to existing FTL routines. As noted above, the major difference is that the FTL tracks the clean/dirty status of each page of data, and the *read* command tests to see if the page is present and can fail if it is not. Hence, a *read* request checks the mapping table in the FTL (described in Section 3.3.2) to see if the page is present before fetching the physical location of the data and initiating a flash read. The *write-dirty* and *write-clean* select a physical location to write the data, then update the page map with the new address and a flag indicating if the bit is dirty. Finally, they initiate a flash transfer to write out the new data. We discuss how this data is kept consistent below.

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

3.3.2 Internal Mechanisms

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

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

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

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

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

Address mappings. The FlashTier cache manager addresses the SSC using disk logical block numbers, which the SSC translates to physical flash addresses. However, this unification introduces a high degree of sparseness in the SSC address space, since only a small fraction of the disk blocks are cached within the SSC. The FlashTier design supported sparse address spaces with a memory-optimized data structure based on perfect hash function [14]. This data structure used dynamically allocated memory to grow with the number of mappings, unlike a statically allocated table. However, the OpenSSD firmware has only limited memory management features and uses a slow embedded ARM processor, which makes use of this data structure difficult.

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

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

In the prototype, we maintain a mapping of 4 KB pages, which reduces the cost of garbage collection because pages do not need to be coalesced into larger contiguous blocks. We found this greatly improved performance for random workloads and reduced the cost of garbage collection.

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

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

Name	Function
nameless-write	Write a data block (with no logical address) and returns its assigned physical address
virtual-write	Write a metadata block (with a file system assigned logical address)
physical-read	Read a data block using its physical address
virtual-read	Read a metadata block using its logical address
free	Free, or trim, a block
migration-callback	Up-call into the file system to indicate a data block has been moved to a new physical address

Table 3: **The Nameless Write SSD Interface.**

3.3.3 Lessons Learned

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

- Designs that rely on specific hardware capabilities or features, such as atomic writes, access to out-of-band area on flash, and dynamic memory management for device SRAM/DRAM, should consider the opportunity cost of using the feature, as other services within the SSD may have competing demands for it. For example, the OpenSSD flash controller stores ECC in the OOB area and prohibits its usage by the firmware. Similarly, the limited amount of SRAM requires a lean firmware image with statically linked libraries and necessitates the simpler data structure to store the address mapping.
- Many simulation studies use small erase block sizes, typically a few hundreds of KBs. In contrast, the OpenSSD platform and most commercial SSDs use larger erase block sizes from 1–20 MB to leverage the internal way and channel parallelism. This requires address translation at a finer granularity, which makes it even more important to optimize the background operations such as garbage collection or metadata checkpointing whose cost is dependent on mapping and erase block sizes.

3.4 Nameless Write Implementation

The Nameless-Write interface presented unique implementation challenges because it is a much larger change to SSD operations. Table 3 lists the Nameless Write commands. We focus on the additional challenges that Nameless Writes posed beyond those with SSCs. Our Nameless-Writes prototype is an initial implementation of the design's basic functionality but has not yet been optimized for high performance.

3.4.1 Interfaces

Nameless Writes fundamentally change how an SSD interacts with the OS. The first change is the *nameless-write* command, which passes data but no address, and expects the device to return a physical address or an error indicating the write failed. Passing data without an address is simple, as the firmware can simply ignore the address. However, a write reply message only contains 8 status bits and all other fields are reserved and can not be used to send physical addresses through the ATA interface. On an error return, the device can supply the address of the block in the request that could not be written. This seemed promising as a way to return the physical address. However, the device, the AHCI driver, and the ATA layer interpret errors as catastrophic and thus we could not use errors to return the physical address.

Our second approach was to re-purpose an existing SATA command that *already* returns a 64-bit address. Only one command, *READ_NATIVE_MAX_ADDR*, returns an address. The OS would first send *READ_NATIVE_MAX_ADDR*, to which the Nameless-Write device returns the next available physical address. The OS would then record the physical address, and send the *nameless-write* command with that address.

We found that using two commands for a write raised new problems. First, the *READ_NATIVE_MAX_ADDR* command is an unqueuable command in the SATA interface, so both the ATA layer and the device will flush queued commands and hurt performance. Second, the OS may reorder nameless writes differently than the *READ_NATIVE_MAX_ADDR* commands, which can hurt performance at the device by turning sequential writes into random writes. Worse, though, is that the OS may send multiple independent writes that lie on the same flash virtual page. Because the granularity of file-system blocks (4 KB) is different from internal flash virtual pages (32 KB), the device may try to write the virtual page twice *without* erasing the bock. The second write silently corrupts the virtual page's data.

The *migration-callback* command raised additional problems. Unlike *all* existing calls in the SATA interface, a Nameless-Write device can generate this up-call asynchronously during background tasks such as wear leveling and garbage collection. This call notifies the file system that a block has been relocated and it should update metadata to reflect the new location. We first considered piggybacking this call on responses to other commands, but this raises the same problem of returning addresses described above. Alternatively, the file system could periodically poll for moved data, but this may be too costly given the expected rarity of up-calls.

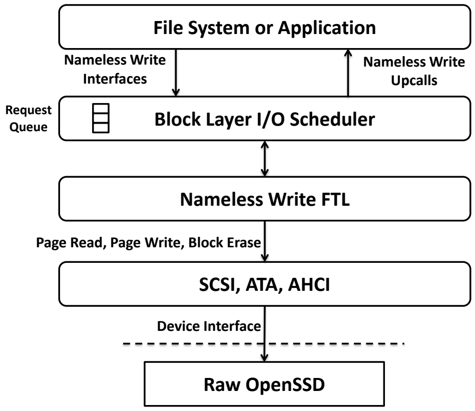
File System or Application
Nameless Write Interfaces
Nameless Write Upcalls
Request Queue
Block Layer I/O Scheduler
Nameless Write FTL
Page Read, Page Write, Block Erase
SCSI, ATA, AHCI
Device Interface
Raw OpenSSD

Figure 6: **Nameless Write FTL.** Implemented as a layer within the host operating system that accesses OpenSSD as a raw flash device.

3.4.2 Internal Mechanisms

Based on the complexity of implementing the full Nameless-Write interface within the device, we opted instead to implement a *split-FTL* design, where the responsibilities of the FTL are divided between firmware within the device and an FTL layer within the host operating system. This approach has been used for PCI-attached flash devices [13], and we extend it to SATA devices as well. In this design, the device exports a low-level interface and the majority of FTL functionality resides as a layer within the host OS.

We built the Nameless-Write FTL at the block layer below the file system and the I/O scheduler and above the SCSI, ATA, and AHCI layers. Figure 6 shows the design. The FTL in the host OS implements the full set of interfaces listed in Table 3, and we implemented a basic firmware that provides *flash-page read*, *flash-page write*, and *flash-block erase*. The host FTL converts a command in the Nameless-Write interface into a sequence of low-level flash operations.

The Nameless-Write FTL processes I/O request queues before they are sent to the lower layers. For each write request queue, the FTL finds a new flash virtual page and assigns physical addresses in the virtual page to the I/Os in the request queue in a sequential order. We change the I/O scheduler to allow a request queue to be at most the size of the flash virtual page (32 KB with OpenSSD); going beyond the virtual page size does not improve write performance but complicates FTL implementation. We choose not to use the same flash virtual page across different write request queues, since doing so will lead to data corruption; lower layers may reorder request queues, resulting in the device write the same virtual page without erasing it. The write performance is highly dependent on the size of the request queues, since each request queue is assigned a flash virtual page; larger request queues result in more sequential writes at the device level. Therefore, to improve random write performance, we change the kernel I/O scheduler to merge any writes (virtual or physical) to the nameless-writing SSD device. We treat virtual writes in a similar way as physical writes. The only difference is that when we assign a physical address to a virtual write, we keep the address mapping in the FTL.

For read requests, we disallow merging of physical and virtual reads and do not change other aspects of the I/O scheduler. For virtual reads, we look up the address mapping in the FTL.

The FTL in the host OS maintains all metadata that were originally maintained by the device firmware, including valid pages, free block list, block erase counts, and bad block list. On a *flush*, used by *fsync()*, the FTL writes all the metadata to the device and records the location of the metadata at a fixed location.

The FTL uses the valid page information to decide which block to garbage collect. It read the valid pages into host DRAM, erases the block, and then writes the data to a new physical block. Once the data has been moved, it sends a *migration-callback* to notify the file system that data has been moved. Because the FTL is in the host OS, this is a simple function call. We note that in this design there is a race condition that can lead to lost data if the system fails before notifying the file system. The proposed Nameless-Write design avoided this race by atomically writing a record of the moved data to the OOB area, but neither atomic writes nor OOB space is available on OpenSSD.

Running the FTL in the kernel provides a hospitable development environment, as it has full access to kernel services. However, it may be more difficult to optimize the performance of the resulting system, as the kernel-side FTL cannot take advantage of internal flash operations, such as copy-backs to efficiently move data within the device. For enterprise class PCI-e devices, kernel-side FTLs following NVM-express specification can implement the block interface directly [25] or use new communication channels based on RPC-like mechanisms [23].

3.4.3 Lessons Learned

The implementation of Nameless Writes with OpenSSD and the ATA interfaces raised several valuable lessons.

- Allowing the host OS to write directly to physical addresses is dangerous, because it cannot guarantee correctness properties such as a flash page is always erased before being rewritten. This is particularly dangerous if the internal write granularity is different than the granularity used by the OS.
- Upcalls from the device to the OS do not fit the existing communication channels between the host and device, and changing the control path for returning values is significantly more difficult than introducing new forward commands or signalling benign errors.
- Building the Nameless-Write FTL at the block layer is simpler than at the device firmware since the block layer has simpler interfaces and interacts with the file system directly.
- With the knowledge of SSD hardware configuration, the kernel I/O scheduler can be changed to improve I/O perfomance with an in-kernel FTL.

4 Evaluation

The overall goal of implementing SSCs and Nameless Writes in a hardware prototype is to validate the design choices. For the SSC design, the prototype is complete to fully benchmark with arbitrary workloads. The Nameless-Write prototype is complete enough to test the performance of the new interface.

4.1 Methodology

We compare the SSC design against a system using only a disk and a system using the optimized OpenSSD as a cache with Facebook's unmodified FlashCache software [10]. We enable *selective caching* in the cache manager, which uses the disk for sequential writes and only sends random writes to the SSC. This feature was already present in Facebook's FlashCache software, upon which we based our cache manager. We evaluate using standard workload profiles with the filebench benchmark.

We use a system with 3 GB DRAM and a Western Digital WD1502FAEX 7200 RPM 1.5 TB plus the OpenSSD board configured with 4 flash modules (128 GB total). We reserve a 75 GB partition on disk for test data, and configure the OpenSSD firmware to use only 8 GB in order to force garbage collection. We compare three systems: *No-Cache* uses only the disk, *SSD* uses unmodified FlashCache software in either write-through (*WT*) or write-back (*WB*) mode running on the optimized baseline SSD (Section 3.1). The *SSC* platform uses our SSC implementation with a version of FlashCache modified to use FlashTier's caching policies, again in write-back or write-through mode.

4.2 SSC Prototype 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) —

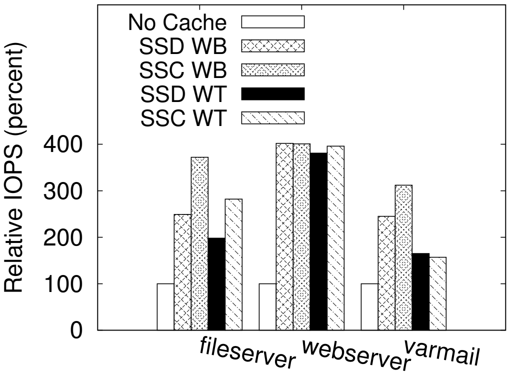

Figure 7: **SSC Prototype Performance.** The performance of write-back and write-through caches using SSD and SSC relative to no-cache disk execution.

using SSD and SSC in write-back and write-through caching modes, similar to FlashTier [32]. The performance is normalized to the *No-Cache* system.

First, we find that both SSD and SSC systems significantly outperform disk for all three workloads. This demonstrates that the platform is fast enough to execute real workloads and measure performance improvements. Second, we find that for the write-intensive fileserver workload, the SSC prototype shows benefits of silent eviction. In the write-back configuration, the SSC performs 52% better than the improved baseline SSD. In the write-through configuration, the SSC performs 45% better. For the read-intensive webserver workload, we find that most data accesses are random reads and there is no garbage collection overhead or silent eviction benefit, which repeats FlashTier projections. In addition, there was little performance loss from moving eviction decisions out of the cache manager and into the FTL.

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

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.

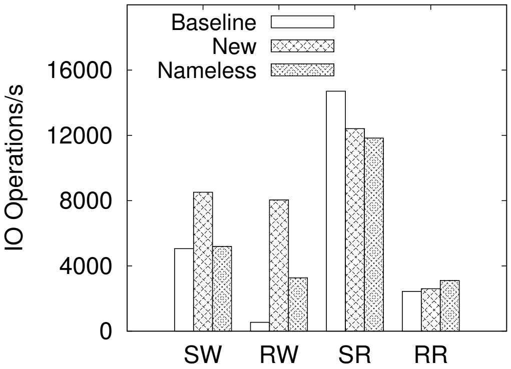

Figure 8: **Read/Write Performance**: On baseline, new (optimized) OpenSSD FTLs, and Nameless split-FTL for fio benchmark with 4 KB request size.

	File System Size				
FTL	4 GB	8 GB	16 GB	32 GB	48 GB
Page-Map	2.50 MB	9.10 MB	17.8 MB	35.1 MB	52.8 MB
Nameless	94 KB	189 KB	325 KB	568 KB	803 KB

Table 4: **FTL Memory Consumption.**

Overall, our results with OpenSSD correlate with the published FlashTier 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%.

4.3 Nameless Writes Prototype

We evaluate the Nameless-Write prototype to validate the performance claims of the interface and memory consumption as projected earlier in simulation [36]. We compare Nameless-Write prototype against the baseline unoptimized and optimized page-mapped FTL. We measure performance with *fio* microbenchmarks and memory consumption with different file system images. We execute the experiments on an OpenSSD board with two flash chips (8 flash banks with 64 GB total).

Figure 8 shows the random (4 KB blocks) and sequential write and read performance with the baseline, optimized, and Nameless-Write FTLs. For sequential writes, sequential reads, and random reads, the Nameless-Write FTL has similar IOPS as the baseline page-mapped FTL. It assigns physical addresses in sequential order, which is the same as the baseline FTL. For random writes, Nameless-Writes perform better than the baseline but worse than sequential write of any FTL. Even though we change the I/O scheduler to merge random writes, we find the random write request queue size is smaller than the size of the flash virtual page.

Table 4 presents the memory usage of the page-mapped FTL (baseline or optimized) and the Nameless-Write FTL with different file system image sizes (from 4 GB to 48 GB). We used Impressions to create these file system images [1]. The memory consumption includes the address mapping tables and all additional FTL metadata. The Nameless-Write FTL uses much less memory than the page-mapped FTL. Unlike the page-mapped FTL, the Nameless-Write FTL does not need to store the address mappings for nameless writes (data) and only stores address mappings for virtual writes (metadata).

Overall, we find that the core Nameless-Write design performs similar to the page-mapped FTL and provides significant memory savings as projected earlier [36]. The major source of performance difference between Nameless-Write and optimized page-mapped FTL is the additional flash optimizations, such as random write buffering, required to overcome the mismatch between hardware and software page sizes.

5 Related Work

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

Hardware Prototypes. Research platforms for characterizing flash performance and reliability have been developed in the past [4, 5, 9, 20, 21]. In addition, there have been efforts on prototyping phase-change memory based prototypes [3, 6]. 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 [34, 35] and educational purposes [8]. Furthermore, we investigated changes to the flash-device interface, while past work looks at internal FTL mechanisms.

New Flash Interfaces. In addition to FlashTier [32] and Nameless Writes [36], there have been commercial efforts on exposing new flash interfaces for file systems [19], caching [11, 18, 24] and key-value stores [12]. However, there is little known to the application developers about the customized communication channels used by the SSD vendors to implement new application-optimized interface. We focus on these challenges and propose solutions to overcome them.

While we re-use the existing SATA protocol to extend the SSD interface, another possibility is to bypass the storage stack and send commands directly to the device. For example, Fusion-io and the recent NVM Express specification [25] attach SSDs to the PCI express bus, which allows a driver to implement the block interface directly if wanted. Similarly, the Marvell DragonFly cache [23] bypasses SATA by using an RPC-like interface directly from a device driver, which simplifies integration and reduces the latency of communication.

6 Discussion and Conclusions

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

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

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

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

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

Acknowledgements

This work is supported by National Science Foundation (NSF) grants CNS 1218485, 1218405, and 1116634, and by a gift from NetApp. We would like to thank Sang-Phil Lim [35] for original FTL implementations on the OpenSSD platform, and our shepherd Steven Swanson for useful feedback. Swift has a financial interest in Microsoft Corp.

References

[1] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating Realistic Impressions for File-System Benchmarking. In *FAST* (2009).

[2] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for ssd performance. In *USENIX* (2008).

[3] AKEL, A., CAULFIELD, A. M., MOLLOV, T. I., GUPTA, R. K., AND SWANSON, S. Onyx: A protoype phase-change memory storage array. In *Hot-Storage* (2011).

[4] BOBOILA, S., AND DESNOYERS, P. Write endurance in flash drives: Measurements and analysis. In *FAST* (2010).

[5] BUNKER, T., WEI, M., AND SWANSON, S. Ming II: A flexible platform for nand flash-based research. In *UCSD TR CS2012-0978* (2012).

[6] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOV, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *IEEE Micro* (2010).

[7] CHEN, F., LUO, T., AND ZHANG, X. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *FAST* (2011).

[8] COMPUTER SYSTEMS LABORATORY, SKKU. Embedded Systems Design Class. http://csl.skku.edu/ICE3028S12/Overview.

[9] DAVIS, J. D., AND ZHANG, L. FRP: a nonvolatile memory research platform targeting nand flash. In *Workshop on Integrating Solid-state Memory into the Storage Hierarchy, ASPLOS* (2009).

[10] FACEBOOK INC. Facebook FlashCache. https://github.com/facebook/flashcache.

[11] FUSION-IO INC. directCache. http://www.fusionio.com/data-sheets/directcache.

[12] FUSION-IO INC. ioMemory Application SDK. http://www.fusionio.com/products/iomemorysdk.

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

[14] GOOGLE INC. Google Sparse Hash. http://goog-sparsehash.sourceforge.net.

[15] GUNAWI, H. S., PRABHAKARAN, V., KRISHNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving file system reliability with i/o shepherding. In *SOSP* (Oct. 2007), pp. 293–306.

[16] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS* (2009).

[17] GUPTA, A., PISOLKAR, R., URGAONKAR, B., AND SIVASUBRAMANIAM, A. Leveraging value locality in optimizing nand flash-based ssds. In *FAST* (2011).

[18] INTEL CORP. Intel Smart Response Technology. http://download.intel.com/design/flash/nand/325554.pdf, 2011.

[19] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: a file system for virtualized flash storage. In *FAST* (2010).

[20] LEE, S., FLEMING, K., PARK, J., HA, K., CAULFIELD, A. M., SWANSON, S., ARVIND, , AND KIM, J. BlueSSD: An open platform for cross-layer experiments for nand flash-based ssds. In *Workshop on Architectural Research Prototyping* (2010).

[21] LEE, S., HA, K., ZHANG, K., KIM, J., AND KIM, J. FlexFS: a flexible flash file system for mlc nand flash memory. In *Usenix ATC* (2009).

[22] MARTIN, B. Inspecting disk io performance with fio. http://www.linuxtoday.com/storage/2008041000526OSHL, Apr. 2008.

[23] MARVELL CORP. Dragonfly platform family. http://www.marvell.com/storage/dragonfly/, 2012.

[24] MESNIER, M., AKERS, J. B., CHEN, F., AND LUO, T. Differentiated storage services. In *SOSP* (2011).

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

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

[27] OH, Y., CHOI, J., LEE, D., AND NOH, S. H. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST* (2012).

[28] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON file systems. In *SOSP* (2005), pp. 206–220.

[29] ROBERT SKYES, OCZ TECHNOLOGY. Critical role of firmware and flash translation layers in solid-state drive design. In *Flash Memory Summit* (2012).

[30] SAMSUNG. K9LCG08U1M NAND Flash Memory Chip.

[31] SAMSUNG. K9PFG08U5M NAND Flash Memory Chip.

[32] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. FlashTier: A lightweight, consistent and durable storage cache. In *EuroSys* (2012).

[33] THE OPENSSD PROJECT. Indilinx Jasmine Platform. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[34] THE OPENSSD PROJECT. Participating Institutes. http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform.

[35] VLDB LAB. SKKU University, Korea. http://ldb.skku.ac.kr.

[36] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. De-indirection for flash-based ssds with nameless writes. In *FAST* (2012).