De-indirection for Flash-based Solid State Drives

by

Yiying Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
in
Computer Sciences

UNIVERSITY OF WISCONSIN-MADISON

2013

Committee in charge:
Prof. Andrea C. Arpaci-Dusseau (Co-chair)
Prof. Remzi H. Arpaci-Dusseau (Co-chair)
Prof. Shan Lu
Prof. Paul Barford
Prof. Jude W. Shavlik

To my parents

Acknowledgements

I would first and foremost extend my whole-hearted gratitude to my advisors, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. Andrea and Remzi are the reason that I had the opportunity for this exceptional Ph.D. journey. To this day, I still remember the moment when they took me as their student and the joy and hope in my heart.

Andrea and Remzi have showed me what systems research is like and how much fun and challenging it can be. Before this journey with them, I had always liked and believed in the beauty of mathematics and theory. My initial interest in systems research happened when I took Remzi's Advanced Operating Systems course, one of the best courses I have ever taken in my student life. Throughout my Ph.D. studies, Remzi and Andrea have given me numerous pieces of priceless advice, ranging from the details of numbers in figures and spaces in text to the broad vision of how to pick research ideas and how to be successful in a research career. They showed me how to enjoy systems research and look at it from a scientific view.

My best moment every week may be the time after my weekly meeting with them. Even though I often walked in his office with tons of results (one time it was over 200 figures in one meeting) and a tiny amount of words, Remzi tolerated it, understood it, and pointed me to the implications of my work, which had never occured to me before. I was also awed by how much details Remzi keep in his mind; he can always point me to the important and interesting problem and the relevant works in the past. Andrea's deep and broad thoughts also amazed me. She can always point out the big picture and guide me in the right direction. Most important, they have the magic of making me feel excited, highly motivated, and confident about my research.

Next, I would like to thank my other thesis-committee members, Shan Lu, Paul Barford, and Jude Shavlik, for their insights, questions, and advice for my research. I would like to thank all my committee members for taking their time to help me improve this dissertation and adjusting their schedule to attend my final defense.

I would also like to extend my gratitude to several other faculty members, who have helped me greatly during my research journey, Mike Swift, Miron Livny, and Jignesh Patel. My special thank goes to Mike Swift, who has patiently given me a lot of valuable advice both on my research work and on research job hunting (even in the last few minutes on his last work day before his sabbatical).

I am fortunate to have a great set of colleagues at UWisconsin who have helped me in various ways, Yupu Zhang, Vijay Chidambaram, Lanyue Lu, Ao Ma, Leo Arulraj, Tyler Harter, Chris Dragga, Zev Weiss, Swami Sundararaman, Sriram Subramanian, Suli Yang, Mohit Saxena, Deepak Ramamurthi, Abhishek Rajimwale, Joe Meehean, and Laxman Visampalli. I would like to especially thank Yupu Zhang for being a quiet yet helping officemate; I can always turn around my chair and get the answers to all kinds of my questions.

I am grateful to have had the opportunity of two great summers: the internship at Microsoft Research-Silicon Valley and the internship at NetApp Corp. I would like to thank the companies as well as my mentors and managers there: Vijayan Prabakaran at Microsoft Research (now at Datrium), and Gokul Soundararajan, Mark Storer, Lakshmi Bairavasundaram (now at Datrium), Sethuraman Subbiah, and Shankar Pasupathy at NetApp. Working with these two groups of people have been a great fun and a rewarding experience for me.

I am very blessed to have the support my friends have given me. I would first like to thank my friend Anupam Dikshit for his kind support. I would also like to thank my two great roommates and best friends, Ramya Olinchindra and Lizhu Qi, for having given me such great accompany in Madison. My other good friends, Jiaxiu He, Jie Liu, Wei Zhang, Linhai Song, Guoliang Jin, Fei Xu, and Xiao Li, have all helped me in different ways.

Finally, I would like to thank my parents, without whose unconditional love and support this Ph.D. would have been meaningless. Even though they have their own doubts and concerns, they have always had faith in me and supported me in pursuing my dream. They cheered with me even for my tinest success and encouraged me when I felt low. The pride and joy in their eyes when seeing me graduate made it all worthwhile. No words can express my gratitude and love to them. I dedicate this whole dissertation to these two most important people in my life.

Abstract

DE-INDIRECTION FOR FLASH-BASED SSDS
Yiying Zhang

Flash-based solid-state drives (SSDs) have revolutionized storage with their high performance. Modern flash-based SSDs virtualize their physical resources with *indirection* to provide the traditional block interface and hide their internal operations and structures. When using a file system on top of a flash-based SSD, the device indirection layer becomes redundant. Moreover, such indirection comes with a cost both in memory space and in performance. Given that flash-based devices are likely to continue to grow in their sizes and in their markets, we are faced with a terrific challenge: *How can we remove the excess indirection and its cost in flash-based SSDs?*

We propose the technique of de-indirection to remove the indirection in flash-based SSDs. With de-indirection, the need for device address mappings is removed and physical addresses are stored directly in file system metadata. By doing so the need for large and costly indirect tables is removed, while the device still has its freedom to control block-allocation decisions, enabling it to execute critical tasks such as garbage collection and wear leveling.

In this dissertation, we first discuss our efforts to build an accurate SSD emulator. The emulator works as a Linux pseudo block device and can be used to run real system workloads. The major challenge we found in building the SSD emulator is to accurately model SSDs with parallel planes. We leveraged several techniques to reduce the computational overhead of the emulator. Our evaluation results show that the emulator can accurately model important metrics for common types of SSDs, which is sufficient for the evaluation of various designs in this dissertation and in SSD-related research.

Next, we present *Nameless Writes*, a new device interface that removes the need for indirection in flash-based SSDs. Nameless writes allow the device to choose the location of a write; only then is the client informed of the *name* (i.e., address) where the block now resides. We demonstrate the effectiveness of nameless writes by porting the Linux ext3 file system to use an emulated nameless-writing device and show that doing so both reduces space and time overheads, thus making for simpler, less costly, and higher-performance SSD-based storage.

We then describe our efforts to implement nameless writes on real hardware. Most research on flash-based SSDs including our initial evaluation of nameless writes rely on simulation or emulation. However, nameless writes require fundamental changes in the internal workings of the device, its interface to the host operating system, and the host OS. Without implementation in real devices, it can be difficult to judge the true benefit of the nameless writes design. Using the OpenSSD Jasmine board, we develop a prototype of the Nameless Write SSD. While the flash-translation layer changes were straightforward, we discovered unexpected complexities in implementing extensions to the storage interface.

Finally, we discuss a new solution to perform de-indirection, the File System De-Virtualizer (*FSDV*), which can dynamically remove the cost of indirection in flash-based SSDs. FSDV is a light-weight tool that de-virtualizes data by changing file system pointers to use device physical addresses. Our evaluation results show that FSDV can dynamically reduce indirection mapping table space with only small performance overhead. We also demonstrate that with our design of FSDV, the changes needed in file system, flash devices, and device interface are small.

Contents

Chapter 1

Introduction

"All problems in computer science can be solved by another level of indirection"
– often attributed to Butler Lampson, who gives credit to David Wheeler

"All problems in computer science can be solved by another level of indirection, but that usually will create another problem"
– David Wheeler

Indirection, a core technique in computer systems, provides the ability to reference an object with another form [82]. Whether in the mapping of file names to blocks or a virtual address space to an underlying physical one, system designers have applied indirection to improve system flexibility, performance, reliability, and capacity for many years. Even within the storage stack, there are many examples of indirection.

File systems are a classic example of adding indirection on top of storage devices to organize data into easy-to-use forms, as well as to provide consistency and reliability [10]. File systems use file and directory structures to organize data; a data block is mapped from file and file offset to a logical block address using the file system *metadata*.

Another example of indirection happens where modern hard disk drives use a modest amount of indirection to improve reliability by hiding underlying write failures [69]. When a write to a particular physical block fails, a hard disk will remap the block to another location on the drive and record the mapping such that future reads will receive the correct data. In this manner, a drive transparently improves reliability without requiring any changes to the client above.

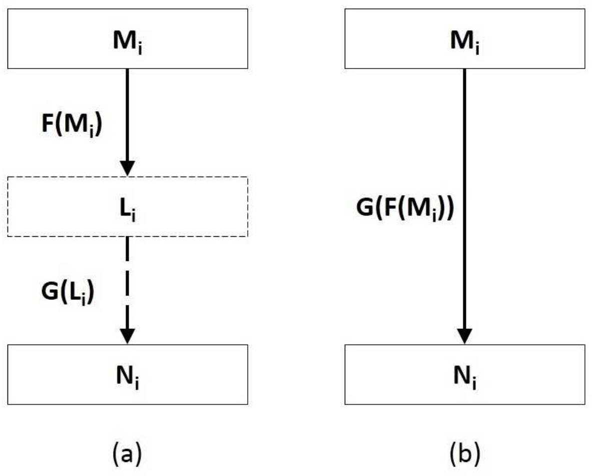


Figure 1.1: **Excess Indirection and De-indirection.** *These graphs demonstrates a system that contains excess indirection (a) and the system after performing de-indirection (b). The dotted part in (a) represents a level of excess indirection, which is removed with de-indirection in (b).*

Because of the benefits and convenience of indirection, system designers often incorporate another level of indirection when designing new systems. As software and hardware systems have become more complex over time (and will continue to do so in the future), layers of indirection have been and will be added. The levels of indirection exist for different reasons, such as providing flexibility and functionality, improving performance, maintaining modularity and code simplicity, and maintaining fixed interfaces.

As a result, *redundant* levels of indirection can exist in a single system, a problem we term *excess indirection*. Specifically, assume that the original form of an object is N_k and its final form (*i.e.*, the form visible by the user) is L_i. If L_i is mapped more than once to transfer into the form of N_k, there are multiple levels of indirection. For example, for two levels of indirection, L_i is first mapped by a function $F(L_i)$ to a form M_j and then mapped by a function $G(M_j)$ to N_k. All together, the mapping for the object is $G(F(L_i)) = N_k$. If one of the levels of indirection can be removed while the system can still function as before, we call this level of indirection redundant and the system has excess indirection. Figure 1.1(a) gives an example of excess indirection.

Unfortunately, indirection comes at a high price, which manifests as performance costs, space overheads, or both. First, mapping tables or some form of metadata are necessary for lookups with indirection. Such metadata requires persistent storage space. Moreover, to improve system performance and reduce the access time to slower storage for the metadata, metadata are often cached in part or in full in fast memory forms like DRAM, creating both monetary and energy costs. There is also performance overhead to access and maintain the indirection metadata.

Excess indirection multiplies the performance and space costs of indirection and is often redundant in a system. It thus presents us a with an important problem of how to reduce the costs of excess indirection.

1.1 Excess Indirection in Flash-based SSDs

Flash memory is a form of non-volatile memory which offers better random-access performance and shock resisdence than traditional hard disks, and lower monetary and energy cost than RAM. Flash memory is often packaged into flash-based *Solid State Devices* (*SSD*s). SSDs have been used as caching devices [18, 29, 52, 61, 67, 74, 81] and hard disk replacements [19, 54, 71, 84], and thus have gained a foothold in both consumer and enterprise markets.

Indirection is particularly important in flash-based SSDs. Unlike traditional storage devices, flash-based SSDs manage an indirection layer and provide a traditional block interface. In modern SSDs, an indirection map in the *Flash Translation Layer* (FTL) allows the device to map writes from the host system's logical address space to the underlying physical address space [24, 34, 40, 48, 59, 60].

FTLs use this indirection for two reasons: first, to transform the erase/program cycle mandated by flash into the more typical write-based interface via copy-on-write techniques, and second, to implement *wear leveling* [47, 51], which is critical to increasing SSD lifetime. Because a flash block becomes unusable after a certain number of erase-program cycles (10,000 or 100,000 cycles according to manufacturers [13, 37]), such indirection is needed to spread the write load across flash blocks evenly and thus ensure that no particularly popular block causes the device to fail prematurely.

The indirection in flash-based SSDs is useful for these purposes. However, flash-based SSDs can exhibit excess indirection. When a file system is running on top of a flash-based SSD, the file system first maps data from file and file offset to logical block addresses; the SSD then maps logical block addresses to device physical addresses. The indirection at the file system level is achieved through the file system metadata; the indirection at the SSD level is achieved through the mapping table in the FTL.

As we can see from the architecture of file system indirection over SSD indirection, there are redundant levels of indirection. Although each level of indirection exists for its own reason (*e.g.*, SSD indirection hides the erase-before-write requirement and the wear-leveling operation), we believe that the indirection in the SSD is redundant and moreover causes memory space and performance cost.

The indirection in SSDs comes with a cost in both memory space and energy. If the FTL can flexibly map each virtual *page* in its address space (assuming a typical page size of 4 KB), an incredibly large indirection table is required. For example, a 1-TB SSD would need 1 GB of table space simply to keep one 32-bit pointer per 4-KB page of the device. There is also a performance cost to maintain and access the mapping tables.

Two trends in flash-based storage make the cost of excess indirection an important issue. First, flash memory is used widely in mobile devices, where energy consumption is a significant concern; a large indirection table in RAM imposes a high energy cost. Second, flash-based storage is gaining popularity in enterprise and cloud storage environments [29, 54]. As the sizes of these flash-based devices scale up, the monetary and energy cost of RAM increases super-linearly. Clearly, a completely flexible mapping is too costly; putting vast quantities of memory (usually SRAM) into an SSD is prohibitive.

Because of this high cost, most SSDs do not offer a fully flexible per-page mapping. A simple approach provides only a pointer per *block* of the SSD (a block typically contains 64 or 128 2-KB pages), which reduces overheads by the ratio of block size to page size. The 1-TB drive would now only need 32 MB of table space, which is more reasonable. However, as clearly articulated by Gupta *et al.* [40], block-level mappings have high performance costs due to excessive garbage collection.

As a result, the majority of FTLs today are built using a hybrid approach, mapping most data at block level and keeping a small page-mapped area for updates [24, 59, 60]. Hybrid approaches keep space overheads low while avoiding the high overheads of garbage collection, at the cost of additional device complexity. Unfortunately, garbage collection can still be costly, reducing the performance of the SSD, sometimes quite noticeably [40]. Regardless of the approach, FTL indirection incurs a significant cost; as SSDs scale, even hybrid schemes mostly based on block pointers will become infeasible.

Recently, approaches have been proposed to dynamically cache a small, hot part of the mapping table in DRAM and the rest of the mapping table in the flash memory itself [40]. Another approach to reduce the cost of indirection in SSDs is to move the indirection layer to the host OS in a software layer [46].

Even with these proposed optimizations to reduce the SSD indirection cost, *excess* indirection still exists when a file system is running on top of a flash-based SSD; a block is first mapped from a file offset to its logical address and then from the logical address to its physical address in the device. Both indirection layers maintain their own address spaces and perform their own address allocation. Space and performance overheads are incurred at both layers to maintain their own lookup structure (*i.e.*, file system metadata and SSD FTL). We can clearly see that there is excess indirection in such a system.

1.2 De-Indirection: Removing Excess Indirection

Because of its high costs, excess indirection presents us with an important problem. One way to reduce the costs of excess indirection is to remove the redundant level(s) of indirection, a technique we call *de-indirection*.

The basic idea of de-indirection is simple. Let us imagine a system with two levels of (excess) indirection. The first indirection F maps items in the L space to items in the M space: $F(L_i) \rightarrow \ M_j$. The second indirection G maps items in the M space to those in the N space: $G(M_j) \rightarrow \ N_k$. To look up the item i, one performs the following "excessive" indirection: $G(F(i))$. De-indirection removes the second level of indirection by evaluating the second mapping $G()$ for all values mapped by $F()$: $\forall\ i : F(i) \leftarrow \ G(F(i))$. Thus, the top-level mapping simply extracts the needed values from the lower level indirection and installs them directly.

There are different ways to perform de-indirection. We identify two methods. The first is to remove the need for one level of indirection completely (*i.e.*, this level of indirection is never created). The second method is to still allow the creation of all levels of indirection, but remove (part of) the indirection periodically or when needed. The former method changes the design of the original system that uses multiple levels of indirection, and thus can involve substantial changes to all layers and the interface between different layers in a system. The second method does not require as much change to the original system, but may not remove as much indirection as the first method.

Notice that even though we propose to remove the redundant level of indirection, we do not want to remove the layer in the system completely (*e.g.*, by combining two layers). In fact, it is one of our major goals to retain the functionality of each layer within itself and to introduce as little change to existing layered systems as possible. We believe that different layers exist for their own reasons;

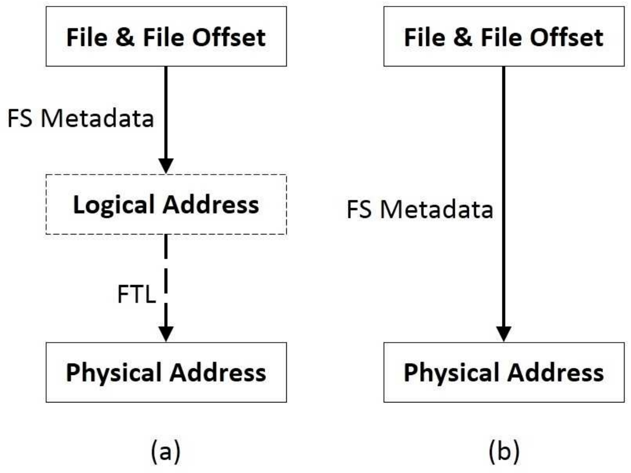


Figure 1.2: **Excess Indirection and De-indirection of Flash-based SSDs.** *The left graph (a) demonstrates the excess indirection in flash-based SSDs when running with a file system. A block is first mapped from file and file offset to the logical address by the file system metadata and is then mapped to the physical address by SSD FTL. The right graph (b) represents the mapping after removing the indirection in the SSD: a block is mapped directly from file and file offset to the physical address using the file system metadata.*

changing them would require significant research and engineering efforts (in each layer) and is less likely to be adopted in real world. Our aim is merely to remove the redundancy in indirection and its associated costs.

De-Indirection of Flash-based SSDs

There are two levels of (redundant) indirection with a file system on top of a flash-based SSD. To remove this excess indirection, we choose to remove the indirection at the SSD level for two reasons. First, the cost of maintaining SSD-level indirection is higher than that of the file system level, since the internal RAM in SSDs incurs a fixed monetary and energy cost and is also restricted by the device's physical size, while main memory is more flexible and is shared by different applications. Second, the file system indirection is used not only with flash-based SSDs, but also other storage devices. It is also used to provide easy-to-use structured data. Thus, removing the file system indirection affects other systems and user applications. With the SSD-level indirection removed, physical block addresses are stored directly in file system metadata; file systems use these addresses for reads and overwrites.

We remove the indirection in SSDs without removing or changing their major functionality, such as physical address allocation and wear leveling. An alternative way to remove indirection is to remove the SSD FTL and manage raw flash memory directly with specific file systems [41, 92, 93]. We believe that exposing raw flash memory to software is dangerous (*e.g.*, a file system can wear out the flash memory either by intention or by accident). Vendors are also less likely to ship raw flash without any wear guarantees.

To perform de-indirection of flash-based SSDs, our first technique is to remove the need for SSD-level address mapping with a new interface called *nameless writes*. This interface removes the need for SSDs to create and maintain its indirection mappings. Our second technique is to have the SSD and file system both create their indirection and operate (largely) unmodified, and to use a tool called "file system de-virtualizer" to occasionally walk through the file system and remove the SSD-level indirection. We next introduce these techniques.

1.3 De-indirection with Nameless Writes

Our first technique to perform de-indirection for flash-based SSDs is a new interface which we term *nameless writes* [95]. With nameless writes, the indirection in flash-based SSDs is directly removed (*i.e.*, the device never creates mappings for the data written with nameless writes).

Unlike most writes, which specify both the *data* to write as well as a *name* (usually in the form of a logical address), a nameless write simply passes the data to the device. The device is free to choose any underlying physical block for the data; after the device *names* the block (i.e., decides where to write it), it informs the file system of its choice. The file system then records the name in its metadata for future reads and overwrites.

One challenge that we encounter in designing nameless writes is the need for flash-based SSDs to move physical blocks for tasks like wear leveling. When the physical address of a block is changed, its corresponding file system metadata also needs to be changed so that the proper block can be found by future reads. Therefore, for physical address changes, we use a new interface called *migration callbacks* from the device to inform the file system about the address changes.

Another potential problem with nameless writes is the recursive update problem: if all writes are nameless, then any update to the file system requires a recursive set of updates up the file-system tree. To circumvent this problem, we introduce a *segmented address space*, which consists of a (large) physical address space for nameless writes, and a (small) virtual address space for traditional named writes. A file system running atop a nameless SSD can keep pointer-based structures in the virtual space; updates to those structures do not necessitate further updates up the tree, thus breaking the recursion.

Nameless writes offer a great advantage over traditional writes, as they largely remove the need for indirection. Instead of pretending that the device can receive writes in any frequency to any block, a device that supports nameless writes is free to assign any physical page to a write when it is written; by returning the true name (i.e., the physical address) of the page to the client above (e.g., the file system), indirection is largely avoided, reducing the monetary cost of the SSD, improving its performance, and simplifying its internal structure.

Nameless writes (largely) remove the costs of indirection without giving away the primary responsibility an SSD manufacturer maintains: wear leveling. If an SSD simply exports the physical address space to clients, a simplistic file system or workload could cause the device to fail rather rapidly, simply by over-writing the same block repeatedly (whether by design or simply through a file-system bug). With nameless writes, no such failure mode exists. Because the device retains control of naming, it retains control of block placement, and thus can properly implement wear leveling to ensure a long device lifetime. We believe that any solution that does not have this property is not viable, as no manufacturer would like to be so vulnerable to failure.

We demonstrate the benefits of nameless writes by porting the Linux ext3 file system to use a nameless SSD. Through extensive analysis on an emulated nameless SSD and comparison with different FTLs, we show the benefits of the new interface, in both reducing the space costs of indirection and improving random-write performance. Overall, we find that compared to an SSD that uses a hybrid FTL (one that maps most SSD area at a coarse granularity and a small area at a fine granularity), a nameless SSD uses a much smaller fraction of memory for indirection while improving performance by an order of magnitude for some workloads.

1.4 Hardware Experience with Nameless Writes

To evaluate our nameless writes design, we built and used an SSD emulator. In the past, most other research on flash-based SSDs also used simulation or emulation for evaluation [6, 74, 78, 40],

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 [43, 30, 32, 73], who release little about the internal workings of their devices. This situation limits the opportunities for research innovation on new flash interfaces, new OS and file system designs for flash, and new internal management software for SSDs.

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 and emulation often simplify systems and may leave out important components, such as the software stack used to access an SSD. For example, our SSD emulator suffers from a few limitations, including a maximum throughput (lowest latency) of emulated device, as will be described in Chapter 3.

Nameless writes require changes to the block interface, flash management algorithms within the device, the OS storage stack, and the file system. Thus, evaluating nameless writes with emulation alone may miss important issues of nameless writes. Meanwhile, the nameless writes design is an ideal candidate for studying the difference between real hardware and simulation or emulation because of the changes needed by nameless writes at different storage layer. Therefore, we sought to validate our nameless writes design by implementing it as a hardware prototype.

We prototype nameless writes with the OpenSSD Jasmine SSD hardware platform [86]. 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 by our design and evaluation of nameless writes with emulation or in published work on new flash interfaces. 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 hurdles. For example, the I/O scheduler must know which commands can be merged and reordered. Second, we found that returning addresses with a write command is difficult with the ATA protocol, since the normal ATA I/O return path does not allow any additional bits for an address. Third, upcalls from the device to the host file system as required by the migration callbacks turned out to be challenging, since all ATA commands are sent from the host to the device.

To solve these problems, we first tried to integrate the nameless writes interfaces into the SATA interface and implement all nameless writes functionality entirely within the firmware running on the OpenSSD board. However, it turned out that passing data from the device to the host OS through the ATA interface is extremely difficult.

This difficulty 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.

Our evaluation results demonstrate that the nameless writes hardware prototype using the split-FTL design significantly reduces the memory consumption as compared to a page-level mapping FTL, while matching the performance of the page-level mapping FTL, the same conclusion we find with our SSD emulation. Thus, the split-FTL approach may be a useful method of implementing new interface designs relying on upcalls from an SSD to the host.

1.5 A File System De-virtualizer

Nameless writes provide a solution to remove the excess indirection in flash-based SSDs (and the cost of this indirection) by using a new interface between file systems and flash-based SSDs. Specifically, with nameless writes, the file system sends only data and no logical address to the device; the device then allocates a physical address and returns it to the file system for future reads. We demonstrated with both emulation and real hardware that nameless writes significantly reduce both the space and the performance cost of SSD virtualization. However, nameless writes have their own shortcomings.

First, the nameless writes solution requires fundamental changes to the device I/O interface. It also requires substantial changes in the device firmware, the file system, the OS, and the device interface. Our hardware experience with nameless writes demonstrates that they are difficult to integrate into existing systems (see Chapter 5) and may need a complete redesign of the storage stack.

Another problem with nameless writes is that all I/Os are de-virtualized at the same time when they are written. The overhead of nameless writes thus occurs for all writes. However, such overhead caused by de-indirection can be hidden if de-indirection is performed at device idle time and not for all the writes. An emerging type of storage systems maintain the device indirection layer in software [46]. In such case, the indirection mappings do not need to be removed all the time. Using techniques like nameless writes to remove indirection mappings for all the data can turn out to be unnecssary and cause more overhead than needed.

To address the problems with nameless writes, we propose the *File System De-Virtualizer* (*FSDV*), a mechanism to dynamically remove the indirection in flash-based SSDs with small changes to existing systems. The basic technique is simple; FSDV walks through the file system structures and changes file system pointers from logical addresses to physical addresses. Doing so does not require changes in normal I/Os. Unlike nameless writes which requires all I/Os to be de-virtualized, the FSDV tool can be invoked dynamically (for example, when the device is idle). We believe that FSDV provides a simple and dynamic way to perform de-virtualization and can be easily integrated into existing systems.

One major design decision that we made to achieve the goal of dynamic de-virtualization is the separation of different address spaces and block status within a file system. Initially, the file system allocates logical addresses on top of a virtualized device in the traditional way; all blocks are in the *logical address space* and the device uses an indirection table to map them to the *device address space*. FSDV then walks through and de-virtualizes the file system. Afterwards, the de-virtualized contents are in the *physical address space* and corresponding mappings in the device are removed. The file system later allocates and overwrites data for user workloads; the device will add new mappings for these data, causing blocks to be mapped from logical or old physical addresses to current device addresses.

A block thus can be in three states: a mapped logical block, a mapped physical block, or a direct physical block. The first two require indirection mapping entries. When the mapping table space for them is large, FSDV can be invoked to move data from the first two states to the direct state. It can also be invoked periodically or when the device is idle. FSDV thus offers a dynamic solution to remove excess virtualization without any fundamental changes to existing file systems, devices, or I/O interface.

Another design question that we met is related to how we handle the address mapping changes caused by the device garbage collection and wear leveling operations. During these operations, the device moves flash pages to new locations and thus either changes their old mappings or adds new mappings (if they originally were direct physical blocks). In order for FSDV to process and remove the mappings caused by these operations, we choose to associate each block with its inode number and record the inode number if the device moves this block and creates a mapping for it. Later, when FSDV is invoked, it processes the files corresponding to these inode numbers and removes the mappings created because of garbage collection or wear leveling.

We change the write interface to let the file system send the inode number associated with a block when writing it. Though this change is made to a normal I/O interface (something we try to avoid), it only changes the forward direction (from the file system to the device). As will be described in chapter 5, the direction from the device to the file system turns out to be the major difficulty when changing the interface with existing hardware and software stacks. Therefore, we believe that our change to the write interface for FSDV is a viable one; in the future, we plan to explore other options to deal with address mapping changes caused by the device.

We implemented FSDV as a user-level tool and modified the ext3 file system and the SSD emulator for it. The FSDV tool can work with both unmounted and mounted file systems. We evaluate the FSDV prototype with macro-benchmarks and show through emulation that FSDV significantly reduces the cost of device virtualization with little performance overhead. We also found that by placing most of the functionality in FSDV, only small changes are needed in the file system, the OS, the device, and the I/O interface.

1.6 Overview

The rest of this dissertation is organized as follows.

- **Background:** Chapter 2 provides a background on flash memory, flash-based SSDs, file systems, and the ext3 file system.

- **Flash-based SSD Emulator:** Before delving into our solutions to remove excess indirection in flash-based SSDs, we first present in Chapter 3 a flash-based SSD emulator that we built for various flash-related research and the research work in this dissertation. As far as we know, we are the first to build and use an SSD emulator for research work; all previous research uses SSD simulators. We describe the challenges and our solutions to build an accurate and flexible SSD emulator.

- **De-indirection with Nameless Writes:** Our first solution to remove excess indirection in flash-based SSDs is the new interface, nameless writes. Chapter 4 discusses the design of the nameless writes interface and the changes needed in the file system and in the SSD for nameless writes. Nameless writes largely reduce the indirection memory space cost and performance overhead in SSDs.

  Chapter 5 describes our efforts to build nameless writes on real hardware, the challenges that we did not foresee with emulation, and our solutions to them.

- **De-indirection with a File System De-Virtualizer:** Our second solution for de-indirection in flash-based SSDs is the mechanism of a file system de-virtualizer (FSDV). Chapter 6 describes our design of the FSDV mechanism. The FSDV mechanism requires no or few changes to the I/O interface, the OS and file system, and the SSD, yet is able to dynamically reduce the indirection in flash-based SSDs.

- **Related Work:** In Chapter 7, we first discuss systems that exhibit excess indirection and other efforts to perform de-indirection. We then present research work in flash memory and storage interfaces that are related to this dissertation.

- **Conclusions and Future Work:** Chapter 8 concludes this dissertation with a summary of our work, the lessons we have learned, and a discussion of future work.

Chapter 2

Background

This chapter provides a background of various aspects that are integral to this dissertation. First, since we focus on removing the excess indirection in flash-based SSDs, we provide a background discussion on flash memory and flash-based SSDs, their internal structures and the softwares that manage them. We then describe the basics of file systems with a focus on the ext3 file system; we make various changes to ext3 to achieve the goal of de-indirection. Finally, we discuss the block interface between host OSes and block devices and the SATA interface technology which supports the block interface.

2.1 Flash-based SSD: An Indirection for Flash Memory

We now provide some background information on the relevant aspects of NAND flash technology. Specifically, we discuss their internal structure, NAND-flash-based SSDs, and the software that manages them.

2.1.1 Flash Memory and Flash-based SSDs

NAND Flash Memory Internals

Figure 2.1 illustrates the internals of a NAND flash memory cell. A NAND flash memory cell (representing a bit) contains a floating gate (FG) MOSFET [80, 20]. Each gate can store one (SLC) or more (MLC/TLC) bits of information. The FG is insulated from the substrate by the tunnel oxide. After a charge is forced to the FG, it cannot move from there without an external force. Excess electrons can be brought to (program) or removed from (erase) a cell, usually performed by the Fowler-Nordheim (FN) tunneling. After a page is written (programmed),

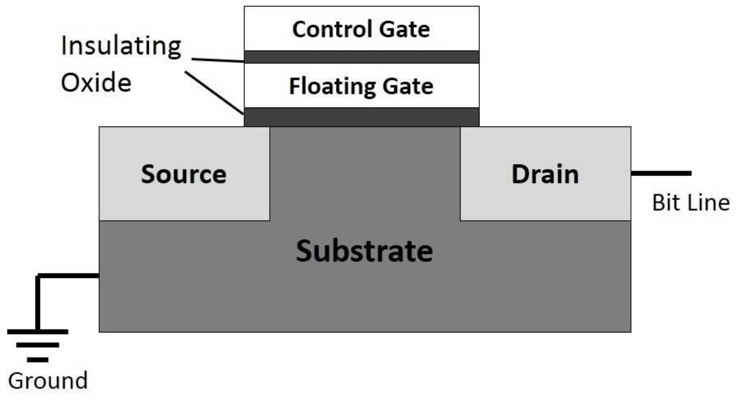


Figure 2.1: **Internals of A Flash Memory Cell.** *This graph illustrates what a NAND flash memory cell looks like. There are two transistor gates (control and floating), which are insulated by thin layers of oxide layer. The number of electrons in the insulated floating gate determines the bit or bits (SLC or MLC/TLC) of the cell. To change the amount of electrons in the floating gate, voltages between the control gate and source or drain are applied.*

it needs to be erased before subsequent writes. Depending on the amount of the charges stored in the FG, a cell can be in two or more logical states. A cell encodes information via voltage levels; thus, being able to distinguish between high and low voltage is necessary to differentiate a 1 from a 0 (for SLC; more voltage levels are required for MLC and TLC) [38]. MLC and TLC are thus denser than SLC but have performance and reliability costs.

NAND flash reads and writes are performed at the granularity of flash *page*, which is typically 2 KB, 4 KB, or 8 KB. Before writing to a flash page, a larger size *erase block* (usually between 64 KB to 4 MB) must be erased, which sets all the bits in the block to 1. Writes (which change some of the 1s to 0s) can then be performed to all the flash pages in the newly-erased block. In contrast to this intricate and expensive procedure, reads are relatively straightforward and can be readily performed in page-sized units.

Writing is thus a noticeably more expensive process than reading. For example, Grupp *et al.* report typical random read latencies of 12 μs (microseconds), write (program) latencies of 200 μs, and erase times of roughly 1500 μs [37]. Thus, in the worst case, both an erase and a program are required for a write, and a write will take more than 100$\times$ longer than a read (141$\times$ in the example numbers above).

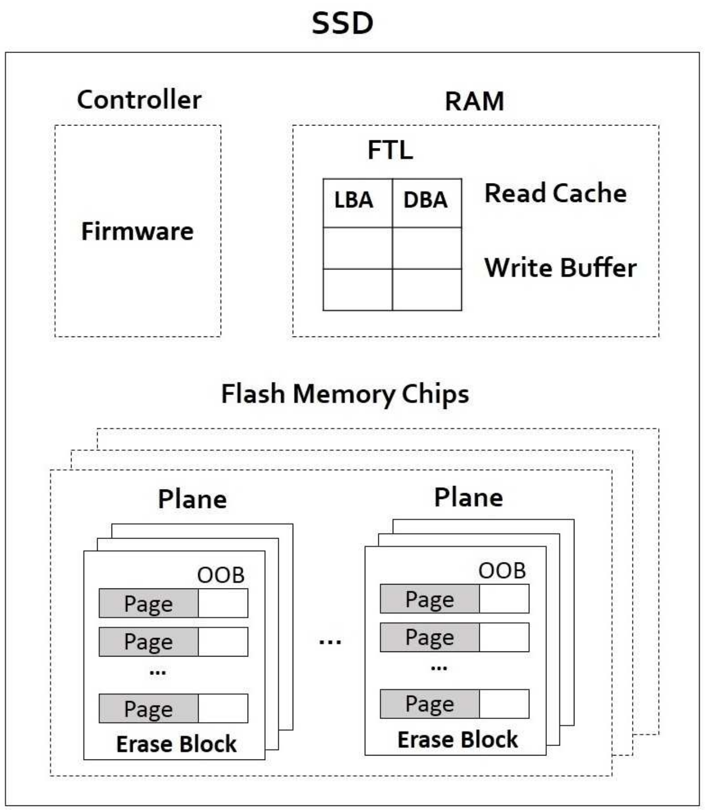


Figure 2.2: **Internals of Flash-based SSD.** *We show the internals of a typical flash-based SSD, which contains a controller, a RAM space, and a set of flash chips.*

An additional problem with flash is its endurance [90]. Each P/E operation causes some damage to the oxide by passing a current through the oxide and placing a high electric field across the oxide, which in turn results in a degradation in threshold voltage. Over time it becomes increasingly difficult to differentiate a 1 from a 0 [1, 13]. Thus, each flash erase block has a lifetime, which gives the number of P/E cycles that the device should be able to perform before it fails. Typical values reported by manufacturers are 100,000 cycles for NAND SLC flash and 10,000 for MLC, though some devices begin to fail earlier than expected [37, 70].

Flash-based SSDs

A *Solid-state drive* (*SSD*) is a storage device that uses (usually non-volatile) memory for data storage. Most modern SSDs use flash memory as their storage medium; early SSDs also use RAM or other similar technology. No hard disks or any device with mechanical moving parts are used in SSDs. Compared to traditional hard disk drives (*HDDs*), SSDs have better performance (especially random performance) and are more shock resistant and quieter. Flash-based SSDs are also cheaper and consume less energy than RAM. Thus, they have gained an increasing foothold in both consumer and enterprise market.

Flash-based SSDs usually contain a set of NAND flash memory chips, an internal processor that runs SSD-management firmware, and a small SRAM and/or DRAM. The processor runs the flash management firmware. The internal RAM is used to store the SSD indirection mapping table and sometimes for a read cache or a write buffer, too. More details about the SSD firmware and mapping tables will be discussed in the next section.

The flash memory chips are used to store data persistently. Flash memory are often organized first into flash pages, then into flash erase blocks, then into planes, dies, and finally into flash chips [6]. An out of band (*OOB*) area is usually associated with each flash page, storing error correction code (*ECC*) and other per-page information. To improve performance (*i.e.*, bandwidth), the flash memory chips are often organized in a way so that multiple flash planes can be accessed in parallel [6]. Figure 2.2 gives an illustration of flash-based SSD internal organization.

Modern NAND flash-based SSDs appear to a host system as a storage device that can be written to or read from in fixed-size chunks, much like modern HDDs. SSDs usually provide a traditional block interface through SATA. In recent years, high-end SSDs start to use the PCIe bus or RPC-like interfaces for better performance and more flexibility [31, 65, 72].

2.1.2 Flash Memory Management Software

For both performance and reliability reasons and to provide the traditional block I/O interface, most flash devices virtualize their physical resources using a *Flash Translation Layer* (FTL) that manages the underlying flash memory and exports the desired disk-like block interface. FTLs serve two important roles in flash-based SSDs; the first role is to improve performance, by reducing the number of erases required per write. The second role is to increase the lifetime of the device through *wear leveling*; by spreading erase load across the blocks of the device, the failure of any one block can be postponed (although not indefinitely).

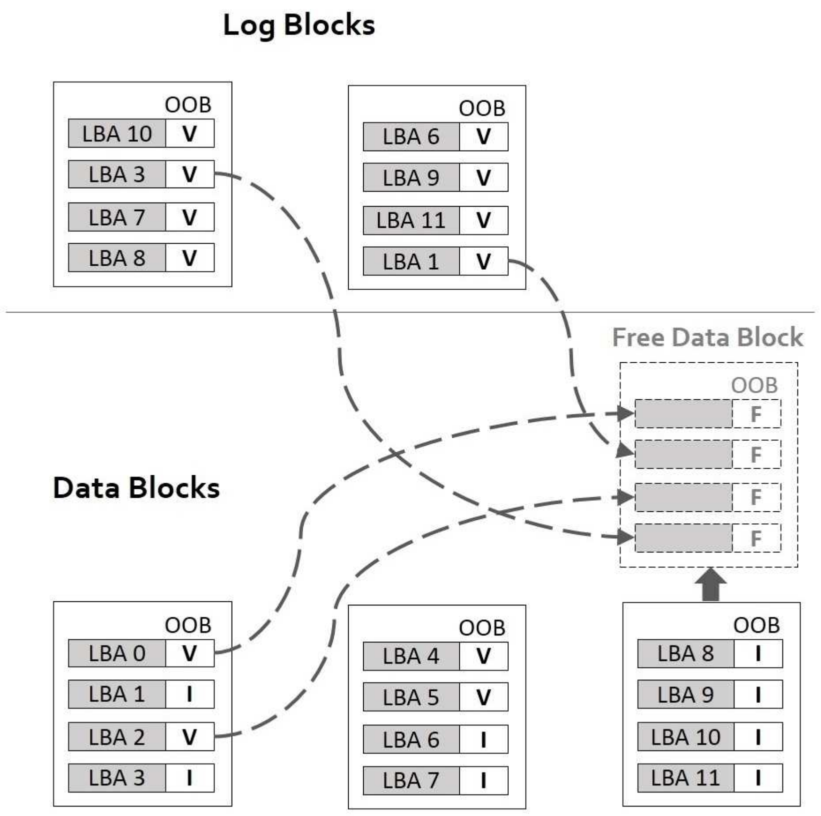


Figure 2.3: **Illustration of a Hybrid FTL before a Merge Operation** *In this example, there are two log blocks and three data blocks in the SSD with a hybrid FTL, each containing four 4 KB flash pages. The data blocks contain 12 pages in total; the SSD thus exposes an effective adddress space of 48 KB (12 $\times$ 4 KB) to the OS. The log blocks are full and a merge operation is triggered. First, to get a free block, the third data block (LBA 8 to 11) is erased, since all its pages are invalid. The FTL then merges logical pages LBA 0 to 3 from their current valid location (two log blocks and one data block) to the erased free data block. After the merge operation, the old data block (leftmost data block) can be erased.*

Both of these roles are accomplished through the simple technique of indirection. Specifically, the FTL maps logical addresses (as seen by the host system) to physical blocks (and hence the name) [42]. Higher-end FTLs never overwrite data in place [35, 40, 59, 60, 62]; rather, they maintain a set of “active” blocks that have

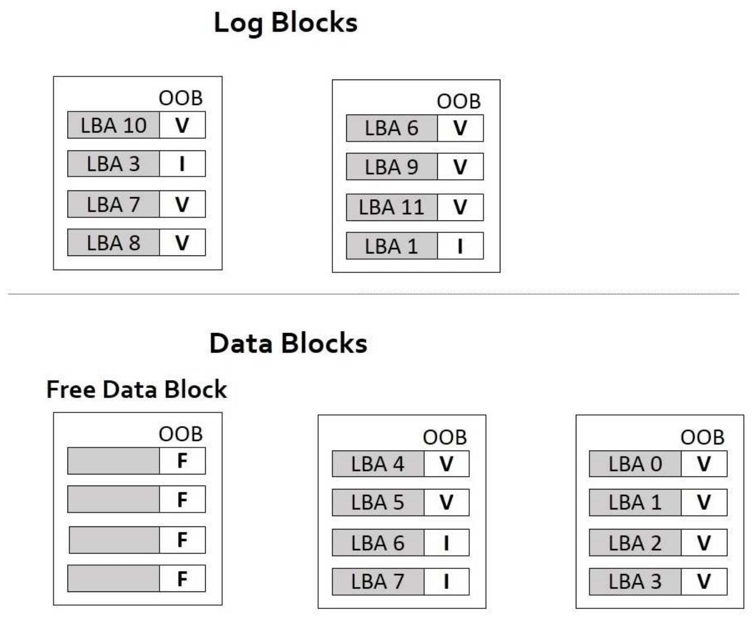


Figure 2.4: **Illustration of a Hybrid FTL after a Merge Operation** *This figure gives an illustration of a typical hybrid FTL after a merge operation. After the merge operation, the old data block (leftmost data block) has been erased and becomes a free block.*

recently been erased and write all incoming data (in page-sized chunks) to these blocks, in a style reminiscent of log-structured file systems [77]. Some blocks thus become "dead" over time and can be garbage collected; explicit cleaning can compact scattered live data and thus free blocks for future usage.

The hybrid FTLs use a coarser granularity of address mapping (usually per 64 KB to 4 MB flash erase block) for most of the flash memory region (*e.g.*, 80% of the total device space) and a finer granularity mapping (usually per 2-KB, 4-KB, or 8-KB flash page) for active data [59, 60]. Therefore, the hybrid approaches reduce mapping table space for a 1 TB SSD to 435 MB, as apposed to 2 GB mapping table space if addresses are mapped all at 4-KB page granularity.

The page-mapped area used for active data is usually called the *log block area*; the block-mapped area used to store data at their final location is called the *data block area*. The pages in a log block can have any arbitrary logical addresses. Log-structured allocation is often used to write new data to the log blocks. The rest of the device is a data block area used to store data blocks at their final locations. The pages in a data block have to belong to the same erase block (*e.g.*, 64 4-KB pages in a 256 KB consecutive logical block address range). The hybrid mapping FTL maintains page-level mappings for the log block area and block-level mappings for the data block area.

When the log block area is full, costly merge operations are invoked. A merge operation is performed to free a data block by *merging* all the valid pages belonging to this data block to a new free block; afterwards the old data block can be erased. Figures 2.3 and 2.4 illustrate the status of a hybrid SSD before and after the merge operation of a data block (which contains LBAs 0 to 3). After this merge operation, two pages in two log blocks are invalidated. To free a log block, all the pages in it need to be merged to data blocks. Thus, such merge operations are costly, especially for random writes, since the pages in a log block can belong to different data blocks. Therefore, some hybrid FTLs maintain a sequential log block for sequential write streams [60]. When the sequential log block is full, it is simply switched with its corresponding data block.

Hybrid FTLs also perform garbage collection for data blocks, when the total amount of free data blocks are low. The merge operations needed to free a data block are the same as described above. To reduce the cost of such merging, a victim data block is often chosen as the data block that has the least amount of valid data.

FTLs also perform *wear leveling*, a technique to extend the life of SSDs by spreading erases evenly across all blocks. If a block contains hot data, it will be written to and erased more often and approaches its lifetime limit faster than blocks with cold data. In such case, a wear leveling operation can simply swap the block containing hot data with a block containing cold data [6]. After this operation, the corresponding mapping table is updated to reflect the new physical addresses of the swapped data–another reason for indirection in flash-based SSDs. In order for the FTL to know the erase cycles and temperature of a block, it needs to keep certain bookkeeping. Most FTLs maintain an erase count with each block. To measure the temperature of a block, a simple technique is to record the time when any of the pages in a block is last accessed. A more accurate method is to record the temperature of all the pages in a block; this method requires more space with each flash page to store the temperature information.

In summary, flash-based SSDs are a type of virtualized storage device, which uses indirection to hide its internal structures and operations and to provide a traditional block I/O interface.

2.2 File System: An Indirection for Data Management

File systems are software systems that organize data and provide an easy-to-use abstract form for storage devices [10].

Most file systems view a storage device as a contiguous *logical address space* and often divide it into fix-sized blocks (e.g. 4 KB). Data blocks are first structured into files; files are then organized into a hierarchy of directories. To manage data with such structures, file systems keep their own metadata, such as block pointers to identify blocks belonging to a file, file size, access rights, and other file properties.

File systems serve as an indirection layer and map data from file and file offsets to logical addresses. A data block is read or written with its file offset. File systems allocate logical block addresses for new data writes. Bitmaps are often used to track the allocation status of the device; a bit represents a logical block address and is set when the block is allocated. Accordingly, file systems perform de-allocation for deletes and truncates, and unset the bit in the bitmap. File systems also perform allocation and de-allocation for file system metadata in a similar way as data allocation and de-allocation. For reads and (in-place) overwrites, file systems look up the file system metadata and locate their logical block addresses. Certain file system metadata (e.g., superblock, root directory block) have fixed locations so that they can always be found without any look-ups.

2.2.1 The Ext3 File System

We now give a brief description of a concrete example of file system, the *ext3* file system. Ext3 is a classic file system that is commonly used in many Linux distributions [10]; thus, we choose to use ext3 in all our works in this dissertation.

A file in ext3 is identified by the structure of *inode* with a unique *inode number*. Ext3 uses a tree structure of pointers to organize data in a file. The inode can be viewed as the tree root, it points to a small number (*e.g.*, twelve) of data blocks. When the file is bigger than this amount of data, the inode also points to an *indirect block*, which in turn points to a set of data blocks. If the file is even bigger, *double* or *triple* indirect blocks are used which points to one or two levels of indirect blocks and eventually to data blocks.

The address space in ext3 is split into block groups, each containing equal size of blocks. Each block group contains a block group descriptor, a data block bitmap, an inode bitmap, an inode table, indirect blocks, and data blocks. Ext3 uses the bitmaps for data and inode allocation and de-allocation. Directories are also stored as files. Each directory contains the information (*e.g.*, file/subdirectory name, inode number) of the files or the subdirectory in the directory. Figure 2.5 illustrates the

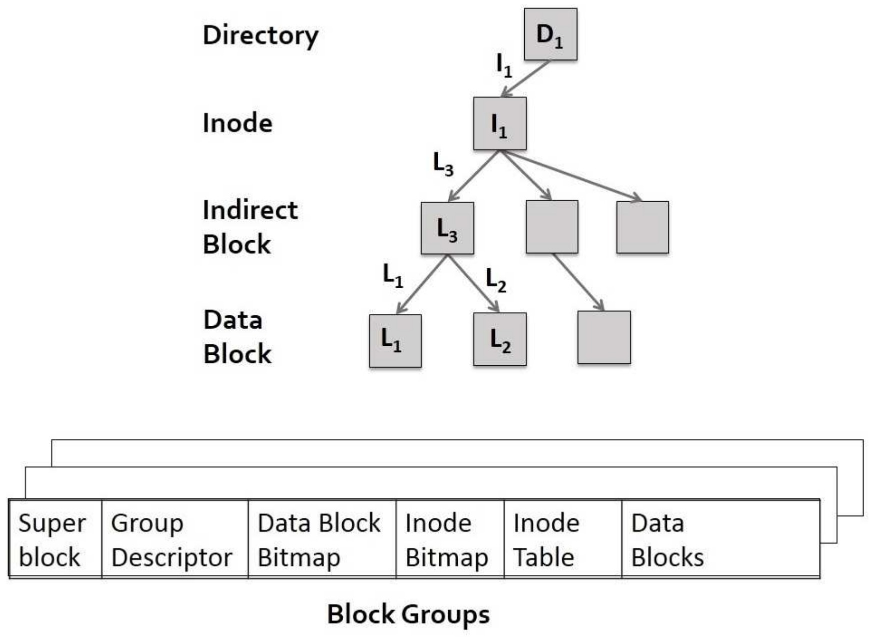


Figure 2.5: **An Illustration of the Ext3 File System.** *This figure shows a simple illustration of the ext3 file system data structures and on-disk layouts. In the top part of the graph, we show the directory and file tree. This example file has one level of indirect blocks and is pointed to directly by the root directory. The bottom part of the graph shows the layout of the ext3 block group.*

directory, file, and block group structures of ext3.

Ext3 also provides reliability and fast recovery through the technique of journaling. Ext3 has three journaling modes: journal mode where both metadata and data are written to the journal, ordered mode where only metadata is journaled but data are guaranteed to be written to disk before metadata in the journal are written, and writeback mode where only metadata is journaled and there is no ordering of metadata and data writes. We choose the ordered mode in the work in this dissertation, since it is a widely used journaling mode.

In summary, file systems such as ext3 organize data into file and directory structures and provide consistency and reliability through the technique of indirection (in the form of file system metadata). File system metadata serve as the means of file system indirection. However, there are certain space and performance cost to maintain this indirection. Combined with the indirection in flash-based SSDs, we see excess indirection with a file system running on top of an SSD.

2.3 Block Interface and SATA

Most flash-based SSDs work as block devices and connect to the OS hosts using the *block interface*. The block interface is also the most common interface for other storage devices such as hard disks. Thus, in this section we give a brief background description of the block interface and a particular hardware interface that supports the block interface: the SATA interface.

The block interface transfers data in the form of fix-sized blocks between a block device and the host OS. All reads and writes use the same block size. The block interface exposes a single address space (the logical address space) and supports sequential and random access to any block addresses. A block I/O is sent from the OS to the device with its block address, a buffer to store the data to be written or read, and the direction of the I/O (*i.e.*, read or write). The OS expects a return from the block interface with the status and error of the I/O and the data to be read.

SATA Interface

The SATA (Serial Advance Technology Attachment) interface is a common interface that works with block devices [85]. It is a replacement for the older PATA (Parallel ATA) interface and uses serial cable for host connection. There are three generations of SATA: SATA 1.0 whose communication rate is 1.5 Gbit/s, SATA 2.0 (3 Gbit/s), and SATA 3.0 (6 Gbit/s).

The SATA interface technology uses layering and contains several layers on both the transmit (host OS) and the receive (device) sides as shown in Figure 2.6. The application and command layers receive commands from the host or the device and then set up and issue commands to the lower layers. The transport layer is responsible for the management of Frame Information Structures (FISes). It formats and passes FISes to the link layer. The link layer converts data into frames and provides frame flow control. Finally, the physical layer performs the actual physical transmission.

ATA commands can be classified into I/O commands and non-data commands. Within I/O commands, there are both PIO (Programmed IO) and DMA (Direct Memory Access) read and write commands. Both PIO and DMA I/O commands have similar forms. The input fields (from host to device) include the command

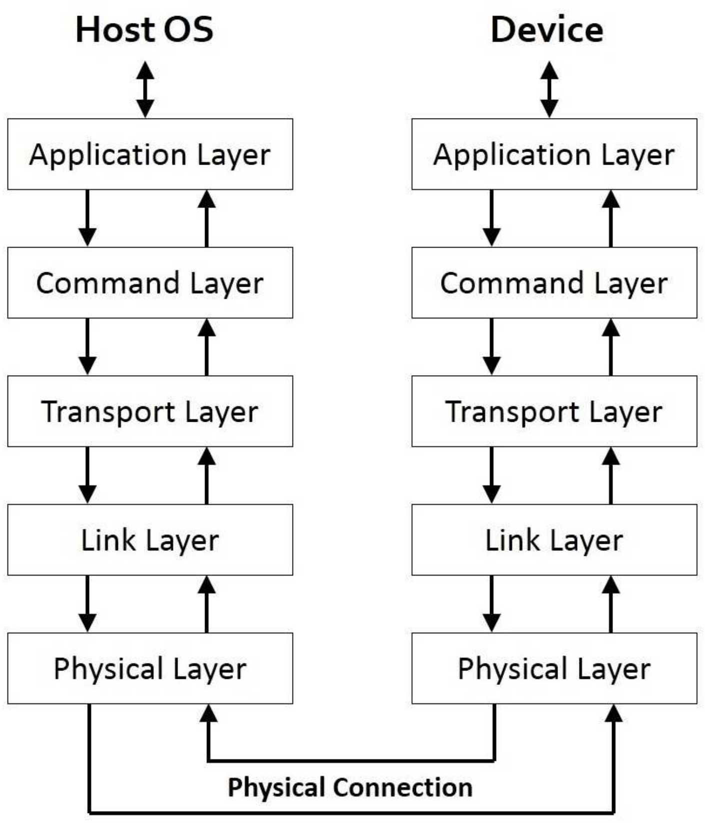


Figure 2.6: **Layered Structure of SATA.** *This graph demonstrates the layers in the SATA technology. On both the host and the device side, there are five layers: application, command, transport, link, and physical layers. The physical layer performs the actual physical communication.*

type, the logical block address and the size of the I/O request, and the device. The return fields (from device to host) include the device, status bits, error bits, and possibly the error LBA. The status bits represent the status of the device (*e.g.*, if busy). For a normal I/O command return, only the status bits are set. The error bits encode the type of error the command encounters. If there is an error (*i.e.*, error bits are set), then the first logical block address where the error occurs is also returned to the host.

The non-data commands in ATA are used for various purposes, such as device configurations, device reset, and device cache flush. The input fields of the non-data commands may include features, LBA, sector count, device, and command. The output fields may include status, error, LBA, and size.

In summary, the block interface is a simple and convenient interface for reading and writing in fixed block size to storage devices. The hardware interface that supports block I/Os is more complex; the SATA interface technology uses multiple layers on both host and device sides. The SATA interface also has a strict set of command protocols and is thus difficult to change.

2.4 Summary

In this chapter, we give different pieces of background for the rest of the dissertation.

We first discuss the technology of flash memory and flash-based SSDs, their internals and the software that controls and manages them. Such software layer uses indirection to hide the internal structures and operations of flash-based SSDs. With this indirection, a block is mapped from its logical address to its physical address. In this process, the SSD software performs allocation in the physical address space.

We then give a brief overview of file systems and a particular example file system, the Linux ext3 file system. The file system is another level of indirection to map a block from its file and file offset to its logical address. The file system performs allocation in the logical address space. Thus, we see redundant levels of address allocation and indirection mappings.

Finally, we describe the interface between the file system and the typical flash-based SSDs: the block interface and the SATA technology.

Chapter 3

A Flash-based SSD Emulator

Because of the increasing prevalence of flash-based SSDs and many unsolved problems of them, a large body of research work has been conducted in the recent years. Most SSD research relies on simulation [6, 40, 74].

Simulation is a common technique to model the behavior of a system. A storage device simulator often takes an I/O event and its arrival time as its input, calculates the time the I/O is supposed to spent with the device, and returns this time as output [15]. A model of a certain device is usually used to calculate the I/O request time. Simulation provides a convenient way to evaluate new design and is relatively easy to implement and debug.

Over the past years, a few SSD simulators have been built and published. The Microsoft Research's SSD simulator is one of the first public SSD simulators and operates as an extension to the DiskSim framework [6]. The PSU FlashSim is another SSD simulator that operates with DiskSim and include several page-level, block-level, and hybrid FTLs [53]. Lee *et al.* built a stand-alone SSD simulator with a simple FTL [56].

These SSD simulators have been used extensively in many SSD research works [6, 40, 22, 96]. However, simulation has its own limitations. First, simulators cannot be used directly to evaluate real workloads on real systems. Real workloads and benchmarks have to be converted specifically for a simulator. In this process of transitioning, different aspects of the workloads and the system can be lost or altered. Second, with simulation, many real system properties and interfaces are simplified. For example, it is difficult to model multithreading behavior with a simulator. Thus, simulation alone is not enough for all evaluation situations and requirements.

Emulation provides another way to evaluate a storage device. A storage device emulator tries to mimic the behavior of a real device. For example, it returns an I/O request at the same wall clock time as what the real device would return. A device emulator also uses a real interface to the host OS. Real workloads and benchmarks can thus run directly on an emulator. Device emulation is thus especially useful to evaluate the interaction of host OS and the device.

Accurate emulation is difficult because of different real system effects, constraints, and non-deterministic nature [36]. The major challenge in implementing an SSD emulator is to accurately model the SSD performance, which is much closer to CPU and RAM than traditional hard disks.

We implemented an SSD emulator and use it throughout different parts of this dissertation. The overall goal of our emulation effort is to be able to evaluate new designs (*e.g.*, SSD de-indirection) using important metrics with common SSD hardware configurations. We leverage several techniques to reduce various computational overhead of the emulator.

Because of the difficulty in building an always-accurate emulator and the limited knowledge of the internals of real SSDs, we do not aim to build an always-accurate emulator that works for all metrics and all SSD configurations. The goal of our emulator is not to model one particular SSD perfectly but to provide insight into the fundamental properties and problems of different types of SSD FTLs. For example, our emulator can accurately emulate write performance for common types of SSDs, but not read performance; writes are the bottleneck to most SSDs and the focus of this dissertation.

As a result, we built a flexible and generally accurate SSD emulator, which works as a block device with the Linux operating system. Our evaluation results show that our SSD emulator has low computational overhead and is accurate for important metrics and common types of SSDs. As far as we know, we are the first to implement an SSD emulator and use it to evaluate new designs.

The rest of this chapter is organized as follows. We first discuss our design and implementation of our SSD emulator in Section 3.1. We then present the evaluation results of the emulator in Section 3.2. Finally, we discuss the limitations of our SSD emulator in Section 3.3 and summarize this chapter in Section 3.4.

3.1 Implementation

We built a flash-based SSD emulator below the Linux block layer; it processes block I/O requests sent from the file system. The SSD emulator has the standard block interface to the host OS and can be used as a traditional block device. Internally, we implemented the emulator as a Linux pseudo block device.

Our goal is to have a generally accurate and flexible SSD emulator so that different SSD and host system designs can be evaluated with real workloads and benchmarks. Since the latency and throughput of modern flash-based SSDs is much closer to those of RAM and CPU than hard disks, the main challenge to build an SSD emulator is to accurately emulate the performance of flash-based SSDs and minimize the computational and other overhead of the emulator. It is especially difficult to emulate the parallelism in a multi-plane SSD. For example, if a single I/O request takes 100μs, then parallel 10 requests to 10 SSD planes will have the effect of finishing 10 requests in 100μs. In the former case (single request), the emulator only needs to finish all its computation and other processing of a request within 100μs, while in the later case, the emulator needs to finish 10 requests in the same amount of time. Thus, reducing the computational overhead of the emulator is important.

We now describe the design and implementation of our SSD emulator and our solution to the challenges discussed above.

3.1.1 Mechanism

We use three main techniques to build an efficient and accurate emulator. First, we store all data in main memory, including file system metadata and data, FTL address mapping table, and other data structures used by the emulator. Second, we separate the data storage and the device modeling operations to two threads. Third, we avoid CPU time as much as possible at the probable cost of memory overhead.

Our original implementation (*Design 1*) used a single thread to perform all tasks, including storing or reading data from memory and calculate request response time by modeling the SSD behavior. Doing so made the total time spent by the emulator for a request higher than the request response time. Therefore, we separate the emulator into two parts in our next design (*Design 2*); each part uses a single thread. The major thread is a thread passed from the kernel with a block I/O request. It first makes a copy of each request and places the copy on a request queue. It then performs data storage for the request. We call this thread the data storage thread. We use another thread that takes request from the request queue, models SSD behavior, and calculates the response time of the request.

The data storage thread is responsible for storing and retrieving data to or from memory. Initially with Design 2, we implemented the data storage thread in a way that for each write, it allocates a memory space, copies the data to be written at the allocated space, and keeps a mapping in a hash table. This implementation turned out to be too costly in computational time. Instead in *Design 3* (our final design), we pre-allocate all the memory space for the emulated device and associate each

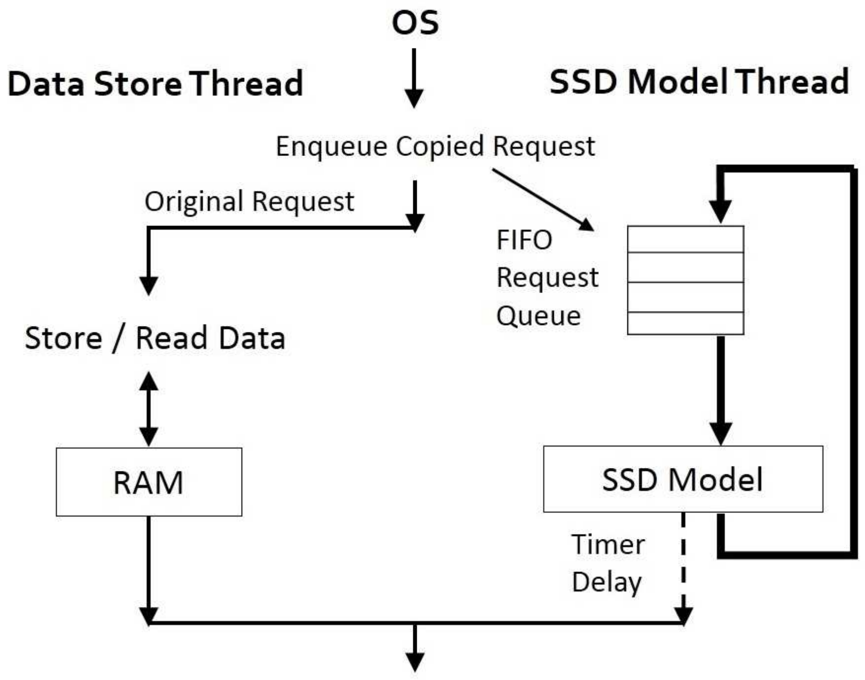


Figure 3.1: **Design of the SSD Emulator.** *This figure describes the basic architecture of the SSD emulator, which contains two threads, the data store thread and the SSD modeling thread. When an I/O request is received from the file system, the main thread makes a copy of it and passes both copies to the two threads to perform memory store/retrieval and SSD simulation. When both threads finish their processing, the request is returned to the file system.*

flash page in the emulated SSD statically with a memory slot (through an array table). Reading and writing thus simply involve an array look-up and memory copy. In this way, no memory allocation or hash table look-up is necessary for each I/O request. Figure 3.1 illustrates the final design of our SSD emulator.

The SSD modeling thread models SSD behavior and maintains a FIFO queue of I/O requests that are passed from the main thread. For each I/O request, its logical block address, its direction, and its arrival time are passed to an SSD simulator. The SSD simulator simulates SSD behavior with a certain FTL and calculates the response time of the request. The SSD model is a separate component and can be replaced by other models. We implemented the SSD simulator based on the PSU objected-oriented SSD simulator codebase [11]. The PSU SSD codebase contains the basic data structures and function skeletons but no implementation of FTLs, address mapping, garbage collection, wear leveling, or I/O parallelism and queueing; we implemented these functionalities. We will discuss more details about our SSD model and its FTLs in the next section.

To accurately emulate the response of an I/O request, we use *hrtimer*, a high-resolution and high-precision Linux kernel timer, to set the time the SSD is supposed to finish the I/O request. When the SSD model returns the response time, if it is larger than the current time, then we set the hrtimer to use this response time. Otherwise, the hrtimer uses the current time (*i.e.*, it expires immediately). The latter case happens when the computation time of the SSD model is larger than the (modeled) latency of an I/O request; in this case, the emulator will not be able to accurately emulate the performance of the modeled SSD. Therefore, it is important to minimize the computation in the SSD model when implementing the SSD FTLs.

An I/O request is considered finished when both the data storage thread and the modeling thread have finished their processing of the I/O. The data storage thread is considered finished with its processing when it has stored or read the I/O data to or from memory. The modeling thread is considered finished when the timer expires. Both threads can return the I/O request to the host, when the other thread and itself have both finished their processing. We maintain an identifier with each I/O to indicate if a thread has finished processing it.

3.1.2 SSD Model

We now discuss our implementation of the SSD simulator and how we model SSD hardware structures and software FTLs. Figure 3.2 illustrates the architecture of the modeled SSD.

We model the SSD internal architecture in the following way. The SSD contains a certain number of packages (flash chips); each package contains a set of dies; each die has a set of planes. A plane is the unit for I/O parallelism. A plane contains a set of flash erase blocks, which are the unit of the erase operation. An erase block contains a set of flash pages, which are the units of reads and writes. There is also an OOB area with each flash page, which is used to store per-page metadata, such as the logical block address of the page and the page valid bit. The logical block address is used to construct the address mapping table during SSD start up and recovery. The valid bit is used during garbage collection and wear leveling. A real SSD also usually stores EEC bits in the OOB area; we do not model the error correcting behavior of the SSD. We also store certain information with each erase

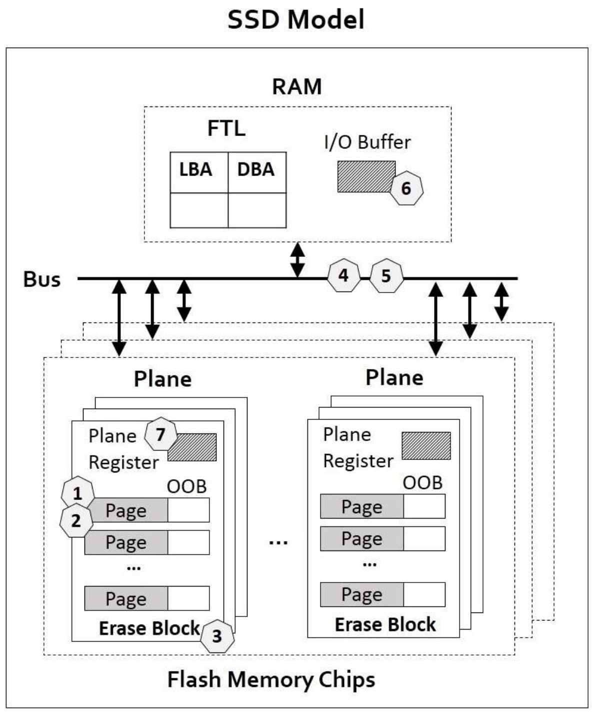


Figure 3.2: **Structures of the SSD Model.** *This figure describes the internal structures of the SSD model. At different circled numbers, there are different types of delay (example values in Table 3.1).*

block, such as the last update time of the block. The update time is used during wear leveling to identify the temperature of the data in an erase block.

We model the SSD firmware with two FTLs, a page-level mapping FTL and a hybrid FTL. Both FTLs make use of multiple planes and parallelize I/O requests to as many planes as possible. To minimize CPU time, we use array table instead of hash table to store all the FTL mapping tables. To reduce computational time, we also maintain a free block queue, so that a full scan is not required to find a new block. We now discuss more details that are specific to each FTL and the garbage collection and wear-leveling operations of both FTLs.

Page-level FTL

The page-level mapping FTL keeps a mapping for each data page between its logical and physical address. For writes, the page-level mapping FTL performs allocation in a log-structured fashion. The FTL maintains an active block in each plane and appends new writes to the next free page in the block. The FTL also parallelizes writes in round-robin order across all the planes. For reads, the page-level mapping FTL simply looks up its mapping table and finds its physical address; it then performs the read from this physical address.

Hybrid FTL

We implemented a hybrid mapping FTL similar to FAST [60], which uses a *log block area* for active data and a *data block area* to store all the data. One sequential log block is dedicated for sequential write streams. All the other log blocks are used for random writes. The rest of the device is a data block area used to store data blocks at their final locations. The pages in a data block have to belong to the same erase block (*e.g.*, 64 4 KB pages in a 256 KB consecutive logical block address range). The pages in a random-write log block can have any arbitrary logical addresses. We choose to use log-structured allocation to write random data to the log blocks. The hybrid mapping FTL maintains page-level mappings for the log block area and block-level mappings for the data block area.

Garbage Collection and Wear Leveling

We implemented a simple garbage collection algorithm and a simple wear-leveling algorithm with both the page-level mapping and the hybrid mapping FTLs.

The garbage collection operation is triggered when the number of free blocks in a plane is low. The garbage collector uses a greedy method and recycles blocks with the least live data. During garbage collection, blocks with all invalid pages are first selected for recycling. The garbage collector simply erases them. The block with the greatest number of invalid pages is then selected. The valid pages in these blocks are written out into a new block. For the page-level mapping FTL, the valid pages are simply written into any free space on any plane (we choose to parallelize these writes to different planes). For the hybrid mapping FTL, to garbage collect a data block, a merge operation is triggered; the valid pages are either copied from the old data block or current log blocks into a new free block.

We implemented a wear-leveling algorithm similar to a previous algorithm [3]. The wear-leveling algorithm is triggered when the overall wear of the SSD is high. The SSD considers both block wear and data temperature during the wear leveling operation. A block whose amount of remaining erase cycles is less than a certain percentage of the average remaining erase cycles of the blocks in the SSD is considered for wear leveling. The SSD then selects the block with the coldest data (oldest update time) and swaps its content with the worm block; the corresponding address mapping entries are also updated accordingly. Notice that because of the need in the wear-leveling algorithm, the SSD also keeps track of the data temperature in each block and stores it with the block.

3.2 Evaluation

In this section, we present our evaluation results of our SSD emulator. We begin our evaluation by answering the questions of how accurate the emulator is and what kind of SSDs the emulator is capable of modeling. Our major goal of the emulator is to have low computational overhead so that it can accurately emulate common types of SSDs with important workloads. After finding out the accuracy and capability of our emulation, we delve into the study of more detailed aspects of the SSD and different FTLs. All experiments are performed on a 2.5 GHz Intel Quad Core CPU with 8 GB memory.

The followings are the specific questions we set to answer.

- What is the computational overhead of the emulator?
- How accurate does the emulator need to be to model an SSD for a particular metric? Can the emulator model common type of SSDs for important metrics?
- What is the effect of multiple parallel planes on the performance of an SSD?
- How does the page-level mapping FTL compare to the hybrid mapping one?
- What is the performance bottleneck of the hybrid mapping FTL and why?

Table 3.1 describes the configurations we used in our evaluation. There are different kinds of latency associated with various SSD internal structures as shown in Figure 3.2. We use two types of flash memory; SSD1 emulates an older generation of flash memory and SSD2 emulates a newer generation.

3.2.1 Emulation Overhead

We first evaluate the computational overhead of our SSD emulator. The computational overhead limits the type of SSDs the emulator can model accurately; if

Configuration	SSD1	SSD2
SSD Size	4 GB	4 GB
Page Size	4 KB	4 KB
Block Size	256 KB	256 KB
Number of Planes *	10	10
Hybrid Log Block Area *	5%	5%
Page Read Latency	25 μs	65 μs
Page Write Latency	200 μs	85 μs
Block Erase Latency	1500 μs	1000 μs
Bus Control Delay	2 μs	2 μs
Bus Data Delay	10 μs	10 μs
RAM Read/Write Delay	1 μs	1 μs
Plane Register Read/Write Delay	1 μs	1 μs

Table 3.1: **SSD Emulator Configurations.** **Number of planes and amount of hybrid log block area use the values in the table as default value but may vary for certain experiments.*

the I/O request time of an SSD is faster than the computation time taken by the emulator, then we cannot model this SSD accurately. Thus, our goal is to have an generally low computational overhead. However, because of the non-deterministic nature of real-time emulation, we allow small amount of outliers.

To evaluate the computational overhead of the emulator, we first study the total time spent in the emulator for an I/O request T_E without including the SSD modeled request time T_R (*i.e.*, $T_R = 0$). Figure 3.3 plots the cumulative distribution of the total emulation time of synchronous sequential and random writes with the page-level mapping and the hybrid mapping FTLs. We use the cumulative distribution, since our goal of the emulator is to have an overall low computational overhead but allow occasional outliers with higher overhead; the cumulative distribution serves to measure if we meet this goal.

Overall, we find that the computational overhead of our emulator is low; the majority of the requests have T_E from 25 to 30 μs for both sequential and random workloads and for both FTLs. Comparing different workloads and FTLs, we find that random writes with the hybrid mapping FTL has higher emulation time than all the others. There is also a long tail, indicating a small number of outliers that have extremely high computational overhead (*e.g.*, 1000 μs). We suspect that the high computational overhead and outliers are due to the complex operations of random writes (*e.g.*, merge and garbage collection) with the hybrid mapping FTL.

To further study the emulator overhead and understand where the bottleneck is,

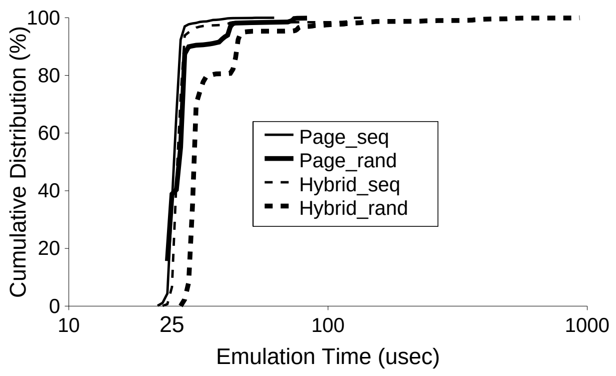


Figure 3.3: **CDF of the Emulation Time.** *We perform sustained synchronous 4 KB sequential and random writes with the page-level mapping and the hybrid mapping FTLs and plot the cumulative distribution of the time spent at the emulator (T_E). The SSD model thread does all its modeling computation but always return a zero modeled request time ($T_R = 0$).*

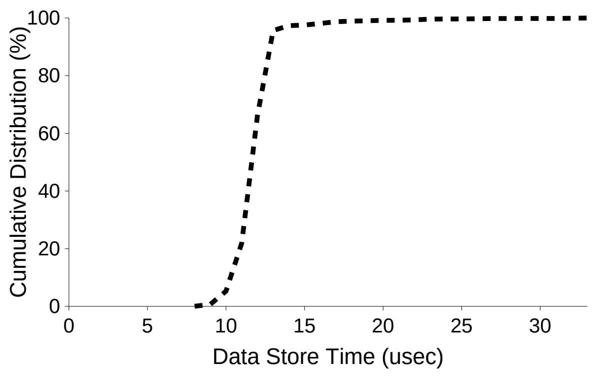


Figure 3.4: **CDF of Data Store Time.** *This figure plots the cumulative density of the time spent at the data store thread using 4 KB sequential writes with the page-level mapping FTL. The results for the hybrid mapping FTL and for random writes are similar to this graph.*

we separately monitor the computational and memory store time taken by the two threads in the emulator to process each I/O request (T_S and T_M). The time taken by the data storage thread includes the time to store or read data from memory and other computational time of the thread. The time taken by the SSD modeling thread includes all the computational time taken by it but not the actual modeled response time of the I/O request ($T_R = 0$).

Figure **??** plots the cumulative distributions of the time taken by the data store thread. The figure plots the distribution of synchronous sequential writes using the page-level mapping FTL, but the other workloads and FTL all have similar distri-

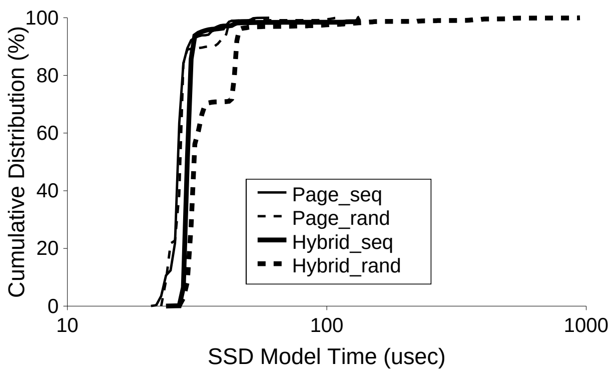


Figure 3.5: **CDF of time spent by the SSD modeling thread for each I/O request.** *We perform sustained synchronous 4 KB sequential and random writes with the page-level mapping and the hybrid mapping FTLs. This figure presents the cumulative density of the computational spent at the SSD modeling thread. The solid line represents the SSD model with the page-level mapping FTL and the dotted line represents the hybrid mapping FTL.*

bution; the data store thread is not affected by the SSD model or type of workloads (as long as the block size in the workloads is the same). We find that the memory store time has a median of 12 μs and is lower than the total emulation time. We also find that the time taken by the data store thread has a small variance, indicating that memory store is a stable operation that is not affected by workloads or types of FTLs.

Figure 3.5 plots the cumulative distributions of the time taken by the SSD modeling thread with sequential and random writes and both the page-level mapping and the hybrid mapping FTLs. We find that the SSD model thread takes a median of 27 to 31μs for the page-level mapping FTL and 29 to 35 μs for the hybrid mapping FTL. The SSD modeling time is similar to the total emulation time and is always bigger than the data store time; the shape of the distributions of the SSD model time is also similar to the distributions of the total emulation time (Figure 3.3). These findings indicate that SSD modeling is the bottleneck of the emulator.

Similar to the total emulation time, we find that random writes with the hybrid mapping FTL has higher modeling time and a longer tail than other workloads and FTL. We find that the computational time of the SSD simulator is higher with merge operation and garbage collection operation; these operations happen more often and have higher cost with the hybrid mapping FTL than the page-level mapping FTL. Fortunately, when an I/O request involves merge or garbage collection operations, its response time is also higher than a request without these operations; therefore, the higher computational overhead is hidden.

3.2.2 Emulation Accuracy and Capability

After knowing the computational overhead of the emulator, we further study the accuracy of the emulator and what types of SSDs it is capable of emulating.

In this set of experiments, we include the simulated SSD request time of a request T_R in the total emulation time T_M. To study the accuracy of the emulator, we set all requests in an experiment to use a fixed SSD request time T_R (*i.e.*, the timer in the emulator is always set to return a request at the time of its arrival time plus T_R). The request time T_R ranges from 0 to 40 μs. Notice that the request time does not include the queueing delay, which the SSD emulator performs before sending a request to the SSD simulator (see Figure 3.1 for the emulator process flow). Also notice that even though we use a fixed SSD request time, we still let the SSD simulator perform its calculation as usual.

Figure 3.6 plots the medians of the emulator time for different T_R's with sequential writes and the page-level mapping FTL. The target line represents the fixed request time we set with the SSD model (T_R). From the figure, we can see that when the simulated request time is equal to or less than 17 μs, the total time spent at the emulator is always around 26 μs, even when the request time is 0 (*i.e.*, the emulator returns a request as soon as it finishes both the data store and SSD modeling computation). When the request time is more than 17 μs, the total time spent at the emulator is always 9 μs longer than the request time. This (fixed) additional time is mainly due to the queueing delay, which is spent before a request goes into the SSD model and thus not included in T_R. Thus, the emulator can accurately model these larger request times, but the lowest request time the emulator can model is 17 μs.

Figure 3.7 plots the medians of the emulator time for different T_Rs with random writes and the page-level mapping FTL. We find that the lowest request time the emulator can model is 19 μs.

Similarly, we perform sequential and random writes with the hybrid mapping FTL and plot the medians of the emulator in Figures 3.8 and 3.9. We find that the lowest request time the emulator can model is 19 μs for sequential writes and 21 μs for random writes.

From the above experiments, we see that the emulator has a limit of a minimum request time that it is capable of emulating accurately because of the computational overhead observed with the emulator (from 17 to 21 μs for different workloads and FTLs).

Since we model SSDs with parallel planes, we must take into consideration the implication of such parallelism on emulation. With parallel planes, the emulator needs to finish processing multiple requests (*i.e.*, one for each plane) in the unit time

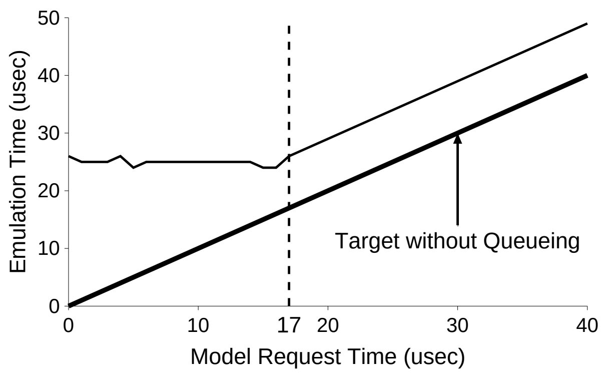


Figure 3.6: **Medians of Emulation Time with Sequential Writes and the page-level mapping FTL.** *The SSD simulator uses a fake fixed modeled request time T_M (0 to 40 μs) for all requests in an experiment. For each experiment, we perform sustained synchronous 4 KB sequential writes and calculate the median of the emulation time T_E. The "Target" line represents the target SSD model request time T_M. This model time does not include any queueing effect. The dotted line represents the minimal SSD request time the emulator can emulate accurately.*

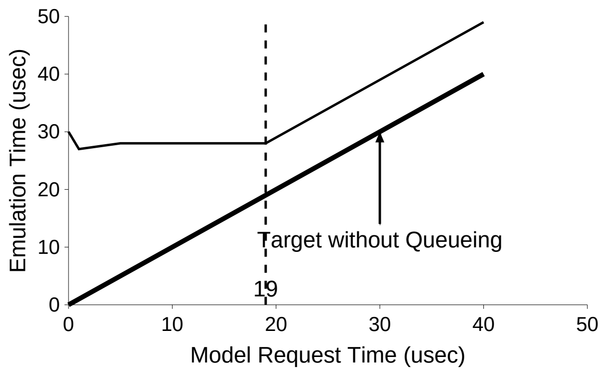


Figure 3.7: **Medians of Emulation Time with Random Writes and the page-level mapping FTL.** *The SSD simulator uses a fake fixed modeled request time T_M (0 to 40 μs) for all requests in an experiment. For each experiment, we perform sustained synchronous 4 KB random writes (within a 2 GB range) and calculate the median of the emulation time T_E. The "Target" line represents the target SSD model request time T_M. This model time does not include any queueing effect. The dotted line represents the minimal SSD request time the emulator can emulate accurately.*

of a per-plane request. To demonstrate this limitation, we plot the area of the SSD configuration space where our emulator can accurately model in Figure 3.10 (with sequential writes and the page-level mapping FTL). For an SSD with 10 parallel planes, this emulation limit means that the emulator can only accurately model per-plane request time of 170 μs or higher. The dot in Figure 3.10 represents the

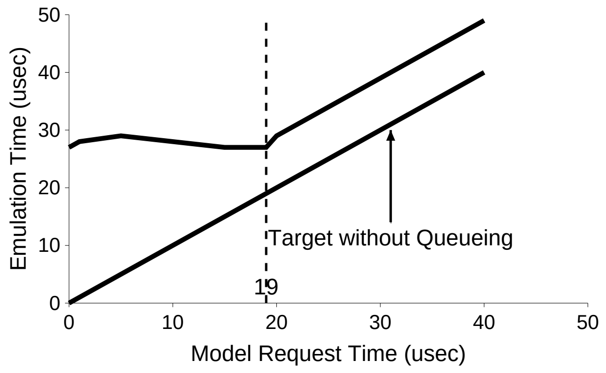


Figure 3.8: **Medians of Emulation Time with Sequential Writes and the hybrid mapping FTL.** *The SSD simulator uses a fake fixed modeled request time T_M (0 to 40 μs) for all requests in an experiment. For each experiment, we perform sustained synchronous 4 KB sequential writes and calculate the median of the emulation time T_E. The "Target" line represents the target SSD model request time T_M. This model time does not include any queueing effect. The dotted line represents the minimal SSD request time the emulator can emulate accurately.*

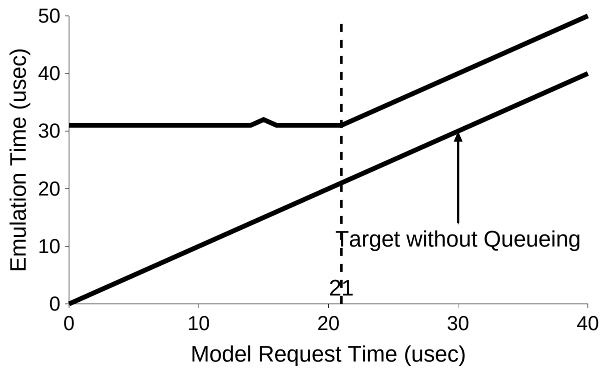


Figure 3.9: **Medians of Emulation Time with Random Writes and the hybrid mapping FTL.** *The SSD simulator uses a fake fixed modeled request time T_M (0 to 40 μs) for all requests in an experiment. For each experiment, we perform sustained synchronous 4 KB random writes (within a 2 GB range) and calculate the median of the emulation time T_E. The "Target" line represents the target SSD model request time T_M. This model time does not include any queueing effect. The dotted line represents the minimal SSD request time the emulator can emulate accurately.*

write configuration we use in later chapters (Chapters 4 and 6). We can see that the write configuration we choose is within the accurate range of the emulator. For other workloads and FTLs, the relationship of number of planes and minimal per-plane request time is similar. For example, the emulator can model per-plane request time of 210 μs with random writes and the hybrid mapping FTL. Notice that

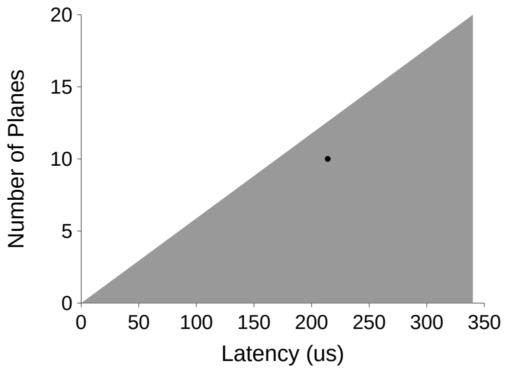


Figure 3.10: **Capability of the Emulator** *This figure illustrates the relationship of number of parallel planes and the minimal per-plane request time that our emulator can model accurately. The grey area represents the configuration space that can be accurately modeled. The black dot represetnts the configuration we choose for evaluation in the rest of this dissertation.*

random writes in real SSDs are also slower than sequential writes with the hybrid mapping FTL, which means that the 210 μs is sufficient for modeling random write performance, too.

For most kinds of flash memory, the unit flash page read time is less than 100 μs. Thus, the emulator is not fit for accurately modeling flash reads with typical number of parallel planes in the SSD. This dissertation (and most other flash research) focus on improving write performance, which is the bottleneck of flash-based SSDs. The emulator can accurately model flash writes (which have larger access time than reads) and is fit for the evaluation in this dissertation.

3.2.3 Effect of Parallelism

I/O parallelism is an important property of flash-based SSDs that enables better performance than a single flash chip. We model the I/O parallelism across multiple planes in our SSD emulator. We now study the effect of such parallelism.

Figure 3.11 shows the throughput of sustained 4 KB random writes with varying number of planes for the page-level mapping and hybrid mapping FTLs. We find that the random write throughput increases linearly with more planes using the page-level mapping FTL for SSD1. The page-level mapping FTL allocates new write across planes in a round robin fashion to parallelize I/Os to all the planes; with more planes, the write throughput is expected to increase. For hybrid mapping FTL, we find that the random write throughput also increases with more planes. However, the effect of increasing planes is not as big as for page-level mapping FTL,

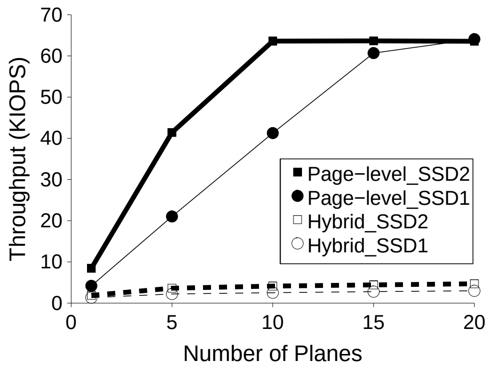


Figure 3.11: **Random Write Throughput with Different Numbers of Planes.** *Throughput of sustained 4 KB random writes with different numbers of planes for Page-Level and Hybrid FTLs. For each data point, we repeat random writes of a 2 GB file until the systems go into a steady state.*

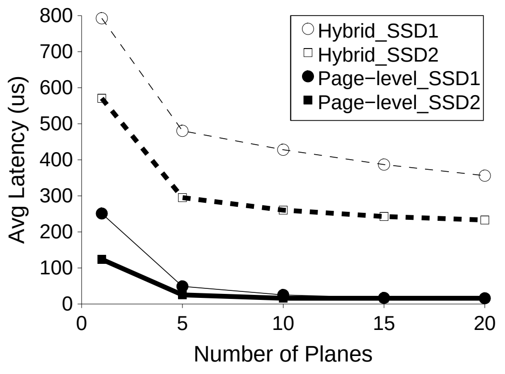


Figure 3.12: **Random Write Avg Latency with Different Numbers of Planes.** *Average latency of sustained 4 KB random writes with different numbers of planes for Page-Level and Hybrid FTLs. For each data point, we repeat random writes of a 2 GB file until the systems go into a steady state.*

since the major reason for poor hybrid mapping random write performance is its costly merge operation and this operation has a high performance cost even with multiple planes.

Looking at SSD2 and the data point from 10 to 20 planes for SSD1, we find that the write throughput of the SSD emulator flattens at around 65 KIOPS. This result conforms with the SSD emulator performance limitation.

We further look at the average request latencies for different number of planes and plot them in Figure 3.12. We find a similar conclusion as the throughput results: the random write average latency decreases with more planes (linearly for the page-level mapping FTL and sub-linearly for the hybrid mapping FTL).

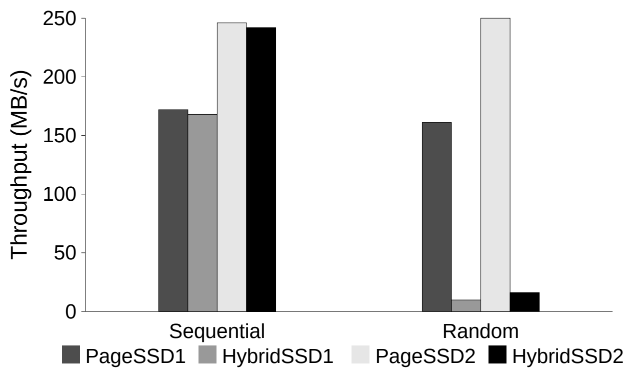


Figure 3.13: **Write Performance of the page-level mapping and the hybrid mapping FTLs.** *All experiments are performed with a 4 GB SSD with 10 parallel planes using two types of flash memory, SSD1 and SSD2. All the experiments use 4 KB block size and are performed in the 4 GB range.*

3.2.4 Comparison of Page-level and Hybrid FTLs

The page-level mapping and hybrid mapping FTLs are two major types of FTLs that differ in the granularity of mappings they maintain (flash page granularity for the page-level mapping FTL and combination of erase block and flash page granularity for the hybrid mapping FTL). We now present the results of comparing the page-level mapping and the hybrid mapping FTLs, considering the mapping table space they use and their write performance.

The mapping table for the 4 GB SSD is 4 MB with the page-level mapping FTL and is 0.85 MB with the hybrid mapping FTL. The page-level mapping FTL keeps a 32-bit pointer for each 4 KB flash page, making the mapping table size 4 MB. The hybrid mapping FTL uses 20% log block area and 80% data block area. For the log block area, it keeps a mapping for each 4 KB page; for the data block area, it keeps a mapping for each 256 KB erase block, making the total mapping table size 0.85 MB. Thus, the page-level mapping FTL uses more mapping table space than the hybrid mapping FTL.

Figure 3.13 presents the write performance results of the page-level mapping and hybrid mapping FTLs. The sequential write throughput for the page-level mapping and hybrid mapping FTLs is similar. But the random write throughput of the hybrid mapping FTL is much lower than the page-level mapping FTL. We also find that as expected, SSD2 has higher write throughput than SSD1 for both page-level mapping and hybrid mapping FTLs, since the flash page write latency is lower for SSD2.

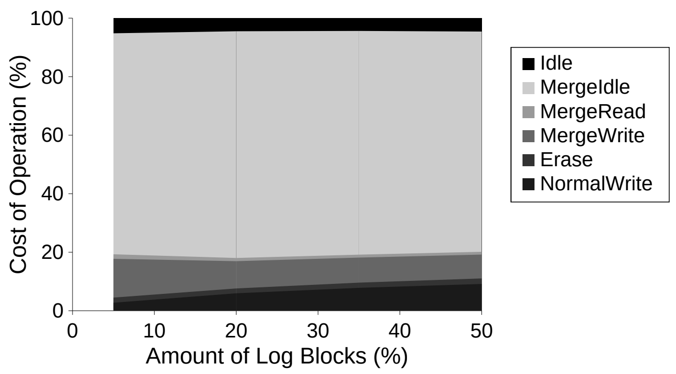


Figure 3.14: **Cost of Different Operations with Hybrid FTL.** *Throughput of sustained 4 KB random writes with different amount of log blocks in Hybrid FTLs. For each data point, we repeat random writes of a 1 GB file until the systems go into a steady state.*

3.2.5 Study of Hybrid FTL

Since random writes are the bottleneck of the hybrid mapping FTL, it is important to learn the cause of its poor random write performance. We now study what factors affect random write performance and the reason for the poor random write performance.

To closely study random write performance and the operations the FTL uses during random writes, we break down the FTL utilization into different operations, including normal writes, block erases, writes and reads during merge operations, idle time during merge operations, and other idle time. Figure 3.14 plots the percentage of all these operations on all the planes during random writes. Figure 3.15 takes a close look at the operations other than the idle time. We find that the merge operations take the majority of the total SSD time and normal writes only take up around 3% to 9% of the total time. As explained in Chapter 2, the merge operation of the hybrid mapping FTL is costly because of the valid data copying and block erases. Surprisingly, with multiple planes, there is also a high idle time during the merge operations. While a data page is copied from a plane during the merge operation, other planes can be idle. As a result, the merge operations are the major reason for poor random writes, even though they are not invoked as often as normal writes (specifically, it is invoked once per 64 4 KB writes to free a 256 KB log block).

Since the costly merges are triggered when the log block area is full, the size of the log block area is likely to affect the random write performance of the hybrid mapping FTL. To study this effect, we change the amount of log blocks in Figures 3.14 and 3.15. We find that with more log blocks, the amount of normal write

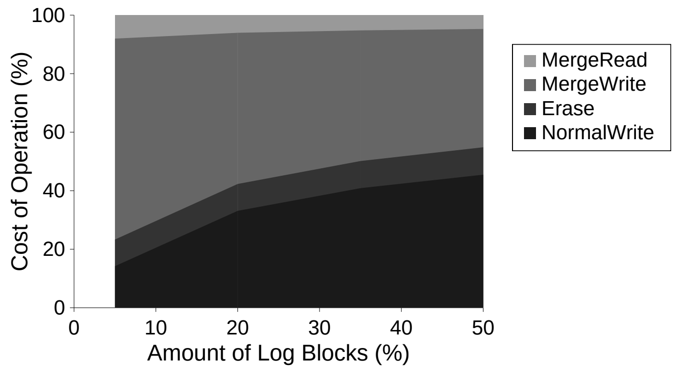


Figure 3.15: **Cost of Different Operations with Hybrid FTL.** *Throughput of sustained 4 KB random writes with different amount of log blocks in Hybrid FTLs. For each data point, we repeat random writes of a 1 GB file until the systems go into a steady state.*

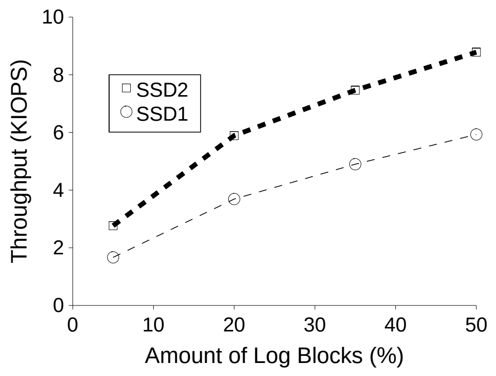


Figure 3.16: **Random Write Throughput with Different Amount of Log Blocks.** *Throughput of sustained 4 KB random writes with different amount of log blocks in Hybrid FTLs. For each data point, we repeat random writes of a 1 GB file until the systems go into a steady state.*

operation time increases and the cost of merge operation is lower. With more log blocks, there is more space for active data. The merge operation cost is lower, since more data will be merged from the log block area and less data will be merged from the data block area; the latter is costlier than the former.

Figure 3.16 plots the random write throughput of the hybrid mapping FTL with different sizes of log blocks. We can see that the random write throughput increases with more log blocks, conforming with the results in the operation break-down analysis. However, more log blocks requires a larger mapping table; for example, 50% log blocks require a 2.03 MB mapping table, while 20% log blocks only require a 0.85 MB mapping table.

3.3 Limitations and Discussions

There are certain limitations with our SSD emulator. In this section we discuss these limitations.

First, our SSD emulator simplifies certain aspects in a real SSD. For example, we do not emulate the write buffer and the read cache available in many modern SSDs. The interface between the SSD emulator and the host OS is also simplified to using the Linux kernel block I/O request interface. In reality, SSDs usually use the SATA interface to connect to the host OS and the SATA interface is more complicated and restricted than the kernel block I/O interface.

Second, we model SSD FTLs based on previous publications and not on real SSDs since there is no public information about details of commercial SSD internals. Reverse engineering commercial SSDs may be a viable solution to learn their internal FTLs. However, our initial effort to reverse engineer a commercial SSD turns out to be difficult; we leave it for future work.

Third, even though we use different techniques to reduce the computational overhead of the emulator, such overhead still limits the speed of the emulator and thus the type of SSDs it can emulate accurately. For example, the emulator cannot emulate a fast flash-based SSD with many parallel planes. For the same reason, it is difficult to emulate a DRAM-based SSD or other fast devices with our emulator. One possible way to alleviate the computational limitation of the emulator is to parallelize the SSD simulator computation across multiple CPU cores.

Finally, our SSD emulator is implemented as a Linux pseudo block driver and does not work with other operating systems.

3.4 Summary

In this chapter, we described the SSD emulator that we built for evaluation of various designs in this dissertation and in other research on SSDs. The emulator works as a pseudo block device with Linux. Workloads and applications can run easily with the emulator in a real system. The SSD model we use in the emulator simulates different components and operations of typical modern flash-based SSDs. We implemented two FTLs for the emulator: the page-level mapping FTL and the hybrid mapping FTL.

A major challenge we found in the process of building the emulator is the difficulty in making the emulator accurate with SSD models that use internal parallism. We used several different techniques to reduce the computational overhead of the emulator so that it can accurately model important types of metrics with common types of SSDs, even those with internal parallism.

Our evaluation results show that the emulator is reasonably fast and can accurately emulate most SSD devices. We further study the page-level mapping and the hybrid mapping FTLs and found that the hybrid mapping FTL has poor random write performance, mainly because of the costly merge operations it uses to free new blocks.

Chapter 4

De-indirection with Nameless Writes

When running a file system on top of a flash-based SSD, excess indirection exists in the SSD and creates both memory space and performance overhead. One way to remove such redundant indirection is to remove the need for the SSD to create and use indirection. Such a goal can be achieved by changing the I/O interface between the file system and the SSD.

In this chapter, we introduce nameless writes, a new I/O interface to remove the costs of the indirection in flash-based SSDs [9, 97]. A nameless write sends only data and no name (*i.e.*, logical block address) to the device. The device then performs its allocation and writes the data to a physical block address. The physical block address is then sent back to the file system by the device and the file system records it in its metadata for future reads.

In designing the nameless writes interface, we encountered two major challenges. First, flash-based SSDs migrate physical blocks because of garbage collection and wear leveling; the file system needs to be informed about such address change so that future reads can be directed properly. Second, if we use nameless writes as the only write interface, then there will be a high performance cost and increased engineering complexity because of the behavior of recursive updates along the file system tree.

We solve the first problem with a *migration callback*, which informs physical address changes to the file system, which then updates its metadata to reflect the changes. We solve the second problem by treating file system metadata and data differently and use traditional writes for metadata and nameless writes for data; the physical addresses of metadata thus do not need to be returned or recorded in the file system, stopping the recursive updates.

We built an emulated nameless-writing SSD and ported the Linux ext3 file system to nameless writes. Our evaluation results of nameless writes and its comparison with other FTLs show that a nameless-writing SSD uses much less memory space for indirection and improves random write performance significantly as compared to the SSD with the hybrid FTL.

The rest of this chapter is organized as follows. In Section 4.1 we present the design of the nameless write interface.In Section 4.2, we show how to build a nameless-writing device. In Section 4.3, we describe how to port the Linux ext3 file system to use the nameless-writing interface, and in Section 4.4, we evaluate nameless writes through experimentation atop an emulated nameless-writing device. Finally, we summarizes this chapter in Section 4.5.

4.1 Nameless Writes

In this section, we discuss a new device interface that enables flash-based SSDs to remove a great deal of their infrastructure for indirection. We call a device that supports this interface a *Nameless-writing Device*. Table 4.1 summarizes the nameless-writing device interfaces.

The key feature of a nameless-writing device is its ability to perform nameless writes; however, to facilitate clients (such as file systems) to use a nameless-writing device, a number of other features are useful as well. In particular, the nameless-writing device should provide support for a segmented address space, migration callbacks, and associated metadata. We discuss these features in this section and how a prototypical file system could use them.

4.1.1 Nameless Write Interfaces

We first present the basic device interfaces of *Nameless Writes*: nameless (new) write, nameless overwrite, physical read, and free.

The nameless write interface completely replaces the existing write operation. A nameless write differs from a traditional write in two important ways. First, a nameless write does not specify a target address (*i.e.*, a name); this allows the device to select the physical location without control from the client above. Second, after the device writes the data, it returns a *physical* address (*i.e.*, a name) and status to the client, which then keeps the name in its own structure for future reads.

The nameless overwrites interface is similar to the nameless (new) write interface, except that it also passes the old physical address(es) to the device. The device

Virtual Read	
down:	virtual address, length
up:	status, data
Virtual Write	
down:	virtual address, data, length
up:	status
Nameless Write	
down:	data, length, metadata
up:	status, resulting physical address(es)
Nameless Overwrite	
down:	old physical address(es), data, length, metadata
up:	status, resulting physical address(es)
Physical Read	
down:	physical address, length, metadata
up:	status, data
Free	
down:	virtual/physical addr, length, metadata, flag
up:	status
Migration [Callback]	
up:	old physical addr, new physical addr, metadata
down:	old physical addr, new physical addr, metadata

Table 4.1: **The Nameless-Writing Device Interfaces.** *The table presents the nameless-writing device interfaces.*

frees the data at the old physical address(es) and then performs a nameless write.

Read operations are mostly unchanged; as usual, they take as input the physical address to be read and return the data at that address and a status indicator. A slight change of the read interface is the addition of metadata in the input, for reasons that will be described in Section 4.1.4.

Because a nameless write is an allocating operation, a nameless-writing device needs to also be informed of de-allocation as well. Most SSDs refer to this interface as the *free* or *trim* command. Once a block has been freed (trimmed), the device is free to re-use it.

Finally, we consider how the nameless write interface could be utilized by a typical file-system client such as Linux ext3. For illustration, we examine the operations to append a new block to an existing file. First, the file system issues a nameless write of the newly-appended data block to a nameless-writing device. When the nameless write completes, the file system is informed of its address and can update the corresponding in-memory inode for this file so that it refers to the physical address of this block. Since the inode has been changed, the file system will eventually flush it to the disk as well; the inode must be written to the device with another nameless write. Again, the file system waits for the inode to be written and then updates any structures containing a reference to the inode. If nameless writes are the only interface available for writing to the storage device, then this recursion will continue until a root structure is reached. For file systems that do not perform this chain of updates or enforce such ordering, such as Linux ext2, additional ordering and writes are needed. This problem of recursive update has been solved in other systems by adding a level of indirection (*e.g.*, the inode map in LFS [77]).

4.1.2 Segmented Address Space

To solve the recursive update problem without requiring substantial changes to the existing file system, we introduce a segmented address space with two segments (see Figure 4.1): the *virtual address space*, which uses virtual read, virtual write and free interfaces, and the *physical address space*, which uses physical read, nameless write and overwrite, and free interfaces.

The virtual segment presents an address space from blocks 0 through $V - 1$, and is a virtual block space of size V blocks. The device virtualizes this address space, and thus keeps a (small) indirection table to map accesses to the virtual space to the correct underlying physical locations. Reads and writes to the virtual space are identical to reads and writes on typical devices. The client sends an address and a length (and, if a write, data) down to the device; the device replies with a status message (success or failure), and if a successful read, the requested data.

The nameless segment presents an address space from blocks 0 through $P - 1$, and is a physical block space of size P blocks. The bulk of the blocks in the device are found in this physical space, which allows typical named reads; however, all writes to physical space are nameless, thus preventing the client from directly writing to physical locations of its choice.

We use a virtual/physical flag to indicate the segment a block is in and the proper interfaces it should go through. The size of the two segments are not fixed. Allocation in either segment can be performed while there is still space on the device. A device space usage counter can be maintained for this purpose.

The reason for the segmented address space is to enable file systems to largely reduce the levels of recursive updates that would occur with only nameless writes.

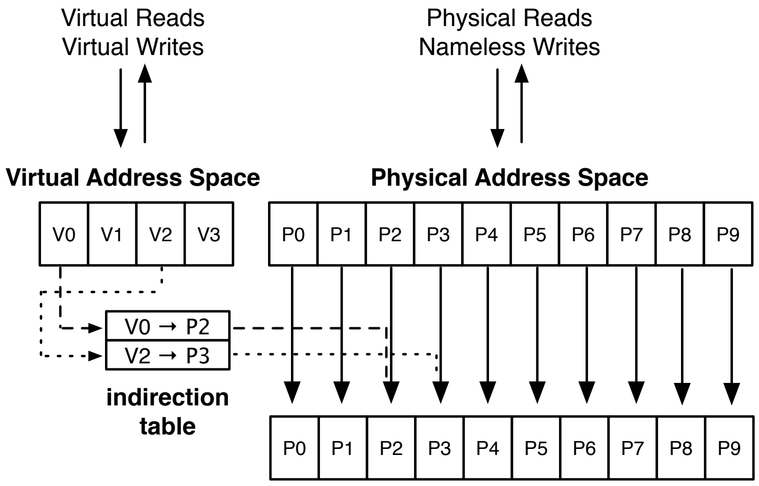


Figure 4.1: **The Segmented Address Space.** *A nameless-writing device provides a segmented address space to clients. The smaller virtual space allows normal reads and writes, which the device in turn maps to underlying physical locations. The larger physical space allows reads to physical addresses, but only nameless writes. In the example, only two blocks of the virtual space are currently mapped, V0 and V2, to physical blocks P2 and P3, respectively.*

File systems such as ext2 and ext3 can be designed such that inodes and other metadata are placed in the virtual address space. Such file systems can simply issue a write to an inode and complete the update without needing to modify directory structures that reference the inode. Thus, the segmented address space allows updates to complete without propagating throughout the directory hierarchy.

4.1.3 Migration Callback

Several kinds of devices such as flash-based SSDs need to migrate data for reasons like wear leveling. We propose the *migration callback* interface to support such needs.

A typical flash-based SSD performs wear leveling via indirection: it simply moves the physical blocks and updates the map. With nameless writes, blocks in the physical segment cannot be moved without informing the file system. To allow the nameless-writing device to move data for wear leveling, a nameless-writing device uses *migration callbacks* to inform the file system of the physical address change of a block. The file system then updates any metadata pointing to this migrated block.

4.1.4 Associated Metadata

The final interface of a nameless-writing device is used to enable the client to quickly locate metadata structures that point to data blocks. The complete specification for associated metadata supports communicating metadata between the client and device. Specifically, the nameless write command is extended to include a third parameter: a small amount of metadata, which is persistently recorded adjacent to the data in a per-block header. Reads and migration callbacks are also extended to include this metadata. The associated metadata is kept with each block buffer in the page cache as well.

This metadata enables the client file system to readily identify the metadata structure(s) that points to a data block. For example, in ext3 we can locate the metadata structure that points to a data block by the inode number, the inode generation number, and the offset of the block in the inode. For file systems that already explicitly record back references, such as btrfs and NoFS [23], the back references can simply be reused for our purposes.

Such metadata structure identification can be used in several tasks. First, when searching for a data block in the page cache, we obtain the metadata information and compare it against the associated metadata of the data blocks in the page cache. Second, the migration callback process uses associated metadata to find the metadata that needs to be updated when a data block is migrated. Finally, associated metadata enables recovery in various crash scenarios, which we will discuss in detail in Section 4.3.7.

One last issue worth noticing is the difference between the associated metadata and address mapping tables. Unlike address mapping tables, the associated metadata is not used to locate physical data and is only used by the device during migration callbacks and crash recovery. Therefore, it can be stored adjacent to the data on the device. Only a small amount of the associated metadata is fetched into device cache for a short period of time during migration callbacks or recovery. Therefore, the space cost of associated metadata is much smaller than address mapping tables.

4.1.5 Implementation Issues

We now discuss various implementation issues that arise in the construction of a nameless-writing device. We focus on those issues different from a standard SSD, which are covered in detail elsewhere [40].

A number of issues revolve around the virtual segment. Most importantly, how big should such a segment be? Unfortunately, its size depends heavily on how the client uses it, as we will see when we port Linux ext3 to use nameless writes in Section 4.3. Our results in Section 4.4 show that a small virtual segment is usually sufficient.

The virtual space, by definition, requires an in-memory indirection table. Fortunately, this table is quite small, likely including simple page-level mappings for each page in the virtual segment. However, the virtual address space could be made larger than the size of the table; in this case, the device would have to swap pieces of the page table to and from the device, slowing down access to the virtual segment. Thus, while putting many data structures into the virtual space is possible, ideally the client should be miserly with the virtual segment, in order to avoid exceeding the supporting physical resources.

Another concern is the extra level of information naturally exported by exposing physical names to clients. Although the value of physical names has been extolled by others [27], a device manufacturer may feel that such information reveals too much of their "secret sauce" and thus be wary of adopting such an interface. We believe that if such a concern exists, the device could hand out modified forms of the true physical addresses, thus trying to hide the exact addresses from clients. Doing so may exact additional performance and space overheads, perhaps the cost of hiding information from clients.

4.2 Nameless-Writing Device

In this section, we describe our implementation of an emulated nameless-writing SSD. With nameless writes, a nameless-writing SSD can have a simpler FTL, which has the freedom to do its own allocation and wear leveling. We first discuss how we implement the nameless-writing interfaces and then propose a new garbage collection method that avoids file-system interaction. We defer the discussion of wear leveling to Section 4.3.6.

4.2.1 Nameless-Writing Interface Support

We implemented an emulated nameless-writing SSD that performs data allocation in a log-structured fashion by maintaining active blocks that are written in sequential order. When a nameless write is received, the device allocates the next free physical address, writes the data, and returns the physical address to the file system.

To support the virtual block space, the nameless-writing device maintains a mapping table between logical and physical addresses in its device cache. When the cache is full, the mapping table is swapped out to the flash storage of the SSD. As our results show in Section 4.4.1, the mapping table size of typical file system images is small; thus, such swapping rarely happens in practice.

The nameless-writing device handles trims in a manner similar to traditional SSDs; it invalidates the physical address sent by a trim command. During garbage collection, invalidated pages can be recycled. The device also invalidates the old physical addresses of overwrites.

A nameless-writing device needs to keep certain associated metadata for nameless writes. We choose to store the associated metadata of a data page in its Out-Of-Band (OOB) area. The associated metadata is moved together with data pages when the device performs a migration.

4.2.2 In-place Garbage Collection

In this section, we describe a new garbage collection method for nameless-writing devices. Traditional FTLs perform garbage collection on a flash block by reclaiming its invalid data pages and migrating its live data pages to new locations. Such garbage collection requires a nameless-writing device to inform the file system of the new physical addresses of the migrated live data; the file system then needs to update and write out its metadata. To avoid the costs of such callbacks and additional metadata writes, we propose *in-place garbage collection*, which writes the live data back to the same location instead of migrating it. A similar hole-plugging approach was proposed in earlier work [66], where live data is used to plug the holes of most utilized segments.

To perform in-place garbage collection, the FTL selects a candidate block using a certain policy. The FTL reads all live pages from the chosen block together with their associated metadata, stores them temporarily in a super-capacitor- or battery-backed cache, and then erases the block. The FTL next writes the live pages to their original addresses and tries to fill the rest of the block with writes in the waiting queue of the device. Since a flash block can only be written in one direction, when there are no waiting writes to fill the block, the FTL marks the free space in the block as unusable. We call such space *wasted space*. During in-place garbage collection, the physical addresses of live data are not changed. Thus, no file system involvement is needed.

Policy to choose candidate block: A natural question is how to choose blocks for garbage collection. A simple method is to pick blocks with the fewest live pages so that the cost of reading and writing them back is minimized. However, choosing such blocks may result in an excess of wasted space. In order to pick a good candidate block for in-place garbage collection, we aim to minimize the cost of rewriting live data and to reduce wasted space during garbage collection. We propose an algorithm that tries to maximize the benefit and minimize the cost of in-place garbage collection. We define the cost of garbage collecting a block to be the total cost of erasing the block (T_{erase}), reading (T_{page_read}) and writing (T_{page_write}) live data (N_{valid}) in the block.

$$cost = T_{erase} + (T_{page\_read} + T_{page\_write}) * N_{valid}$$

We define benefit as the number of new pages that can potentially be written in the block. Benefit includes the following items: the current number of waiting writes in the device queue (N_{wait_write}), which can be filled into empty pages immediately, the number of empty pages at the end of a block (N_{last}), which can be filled at a later time, and an estimated number of future writes based on the speed of incoming writes (S_{write}). While writing valid pages (N_{valid}) and waiting writes (N_{wait_write}), new writes will be accumulated in the device queue. We account for these new incoming writes by $T_{page\_write} * (N_{valid} + N_{wait\_write}) * S_{write}$. Since we can never write more than the amount of the recycled space (i.e., number of invalid pages, $N_{invalid}$) of a block, the benefit function uses the minimum of the number of invalid pages and the number of all potential new writes.

$$benefit = \min(N_{invalid}, N_{wait\_write} + N_{last} \quad (4.1)$$

$$+T_{page\_write} * (N_{valid} + N_{wait\_write}) * S_{write}) \quad (4.2)$$

The FTL calculates the $\frac{benefit}{cost}$ ratio of all blocks that contain invalid pages and selects the block with the maximal ratio to be the garbage collection candidate. Computationally less expensive algorithms could be used to find reasonable approximations; such an improvement is left to future work.

4.3 Nameless Writes on ext3

In this section we discuss our implementation of nameless writes on the Linux ext3 file system with its ordered journal mode. The ordered journaling mode of ext3 is a commonly used mode, which writes metadata to the journal and writes data to disk before committing metadata of the transaction. It provides ordering that can be naturally used by nameless writes, since the nameless-writing interface requires metadata to reflect physical address returned by data writes. When committing metadata in ordered mode, the physical addresses of data blocks are known to the file system because data blocks are written out first.

4.3.1 Segmented Address Space

We first discuss physical and virtual address space separation and modified file-system allocation on ext3. We use the physical address space to store all data blocks and the virtual address space to store all metadata structures, including superblocks, inodes, data and inode bitmaps, indirect blocks, directory blocks, and journal blocks. We use the type of a block to determine whether it is in the virtual or the physical address space and the type of interfaces it goes through.

The nameless-writing file system does not perform allocation of the physical address space and only allocates metadata in the virtual address space. Therefore, we do not fetch or update group bitmaps for nameless block allocation. For these data blocks, the only bookkeeping task that the file system needs to perform is tracking overall device space usage. Specifically, the file system checks for total free space of the device and updates the free space counter when a data block is allocated or de-allocated. Metadata blocks in the virtual physical address space are allocated in the same way as the original ext3 file system, thus making use of existing bitmaps.

4.3.2 Associated Metadata

We include the following items as associated metadata of a data block: 1) the inode number or the logical address of the indirect block that points to the data block, 2) the offset within the inode or the indirect block, 3) the inode generation number, and 4) a timestamp of when the data block is last updated or migrated. Items 1 to 3 are used to identify the metadata structure that points to a data block. Item 4 is used during the migration callback process to update the metadata structure with the most up-to-date physical address of a data block.

All the associated metadata is stored in the OOB area of a flash page. The total amount of additional status we store in the OOB area is less than 48 bytes, smaller than the typical 128-byte OOB size of 4-KB flash pages. For reliability reasons, we require that a data page and its OOB area are always written atomically.

4.3.3 Write

To perform a nameless write, the file system sends the data and the associated metadata of the block to the device. When the device finishes a nameless write and sends back its physical address, the file system updates the inode or the indirect block pointing to it with the new physical address. It also updates the block buffer with the new physical address. In ordered journaling mode, metadata blocks are always written after data blocks have been committed; thus on-disk metadata is always consistent with its data. The file system performs overwrites similarly. The only difference is that overwrites have an existing physical address, which is sent to the device; the device uses this information to invalidate the old data.

4.3.4 Read

We change two parts of the read operation of data blocks in the physical address space: reading from the page cache and reading from the physical device. To search for a data block in the page cache, we compare the metadata index (e.g., inode number, inode generation number, and block offset) of the block to be read against the metadata associated with the blocks in the page cache. If the buffer is not in the page cache, the file system fetches it from the device using its physical address. The associated metadata of the data block is also sent with the read operation to enable the device to search for remapping entries during device wear leveling (see Section 4.3.6).

4.3.5 Free

The current Linux ext3 file system does not support the SSD trim operation. We implemented the ext3 trim operation in a manner similar to ext4. Trim entries are created when the file system deletes a block (named or nameless). A trim entry contains the logical address of a named block or the physical address of a nameless block, the length of the block, its associated metadata, and the address space flag. The file system then adds the trim entry to the current journal transaction. At the end of transaction commit, all trim entries belonging to the transaction are sent to the device. The device locates the block to be deleted using the information contained in the trim operation and invalidates the block.

When a metadata block is deleted, the original ext3 de-allocation process is performed. When a data block is deleted, no de-allocation is performed (i.e., bitmaps are not updated); only the free space counter is updated.

4.3.6 Wear Leveling with Callbacks

When a nameless-writing device performs wear leveling, it migrates live data to achieve even wear of the device. When such migration happens with data blocks in the physical address space, the file system needs to be informed about the change of their physical addresses. In this section, we describe how the nameless-writing device handles data block migration and how it interacts with the file system to perform *migration callbacks*.

When live nameless data blocks (together with their associated metadata in the OOB area) are migrated during wear leveling, the nameless-writing device creates a mapping from the data block's old physical address to its new physical address and stores it together with its associated metadata in a *migration remapping table* in the device cache. The migration remapping table is used to locate the migrated physical address of a data block for reads and overwrites, which may be sent to the device with the block's old physical address. After the mapping has been added, the old physical address is reclaimed and can be used by future writes.

At the end of a wear-leveling operation, the device sends a migration callback to the file system, which contains all migrated physical addresses and their associated metadata. The file system then uses the associated metadata to locate the metadata pointing to the data block and updates it with the new physical address in a background process. Next, the file system writes changed metadata to the device. When a metadata write finishes, the file system deletes all the callback entries belonging to this metadata block and sends a response to the device, informing it that the migration callback has been processed. Finally, the device deletes the remapping entry when receiving the response of a migration callback.

For migrated metadata blocks, the file system does not need to be informed of the physical address change since it is kept in the virtual address space. Thus, the device does not keep remapping entries or send migration callbacks for metadata blocks.

During the migration callback process, we allow reads and overwrites to the migrated data blocks. When receiving a read or an overwrite during the callback period, the device first looks in the migration remapping table to locate the current physical address of the data block and then performs the request.

Since all remapping entries are stored in the on-device RAM before the file system finishes processing the migration callbacks, we may run out of RAM space if the file system does not respond to callbacks or responds too slowly. In such a case, we simply prohibit future wear-leveling migrations until file system responds and prevent block wear-out only through garbage collection.

4.3.7 Reliability Discussion

The changes of the ext3 file system discussed above may cause new reliability issues. In this section, we discuss several reliability issues and our solutions to them.

There are three main reliability issues related to nameless writes. First, we maintain a mapping table in the on-device RAM for the virtual address space. This table needs to be reconstructed each time the device powers on (either after a normal power-off or a crash). Second, the in-memory metadata can be inconsistent with the physical addresses of nameless blocks because of a crash after writing a data block and before updating its metadata block, or because of a crash during wear-leveling callbacks. Finally, crashes can happen during in-place garbage collection, specifically, after reading the live data and before writing it back, which may cause data loss.

We solve the first two problems by using the metadata information maintained in the device OOB area. We store logical addresses with data pages in the virtual address space for reconstructing the logical-to-physical address mapping table. We store associated metadata, as discussed in Section 4.1.4, with all nameless data. We also store the validity of all flash pages in their OOB area. We maintain an invariant that metadata in the OOB area is always consistent with the data in the flash page by writing the OOB area and the flash page atomically.

We solve the in-place garbage collection reliability problem by requiring the use of a small memory backed by battery or super-capacitor. Notice that the amount of live data we need to hold during a garbage collection operation is no more than the size of an SSD block, typically 256 KB, thus only adding a small monetary cost to the whole device.

The recovery process works as follows. When the device is started, we perform a whole-device scan and read the OOB area of all valid flash pages to reconstruct the mapping table of the virtual address space. If a crash is detected, we perform the following steps. The device sends the associated metadata in the OOB area and the physical addresses of flash pages in the physical address space to the file system. The file system then locates the proper metadata structures. If the physical address in a metadata structure is inconsistent, the file system updates it with the new physical address and adds the metadata write to a dedicated transaction. After all metadata is processed, the file system commits the transaction, at which point the recovery process is finished.

4.4 Evaluation

In this section, we present our evaluation of nameless writes on an emulated nameless-writing device. Specifically, we focus on studying the following questions:

- What are the memory space costs of nameless-writing devices compared to other FTLs?

Configuration	Value
SSD Size	4 GB
Page Size	4 KB
Block Size	256 KB
Number of Planes	10
Hybrid Log Block Area	5%
Page Read Latency	25 μs
Page Write Latency	200 μs
Block Erase Latency	1500 μs
Bus Control Delay	2 μs
Bus Data Delay	10 μs
RAM Read/Write Delay	1 μs
Plane Register Read/Write Delay	1 μs

Table 4.2: **SSD Emulator Configuration.** *This table presents the configuration we used in our SSD emulator. The components for each configuration can be found in Figure 3.2*

- What is the overall performance benefit of nameless-writing devices?
- What is the write performance of nameless-writing devices? How and why is it different from page-level mapping and hybrid mapping FTLs?
- What are the costs of in-place garbage collection and the overheads of wear-leveling callbacks?
- Is crash recovery correct and what are its overheads?

We implemented the emulated nameless-writing device with our SSD emulator described in Chapter 3. We compare the nameless-writing FTL to both page-level mapping and hybrid mapping FTLs. We implemented the emulated nameless-writing SSD and the nameless-writing ext3 file system on a 64-bit Linux 2.6.33 kernel. The page-level mapping and the hybrid mapping SSD emulators are built on an unmodified 64-bit Linux 2.6.33 kernel. All experiments are performed on a 2.5 GHz Intel Quad Core CPU with 8 GB memory.

4.4.1 SSD Memory Consumption

We first study the space cost of mapping tables used by different SSD FTLs: nameless-writing, page-level mapping, and hybrid mapping. The mapping table size of page-level and hybrid FTLs is calculated based on the total size of the device, its block

Image Size	Page	Hybrid	Nameless
328 MB	328 KB	38 KB	2.7 KB
2 GB	2 MB	235 KB	12 KB
10 GB	10 MB	1.1 MB	31 KB
100 GB	100 MB	11 MB	251 KB
400 GB	400 MB	46 MB	1 MB
1 TB	1 GB	118 MB	2.2 MB

Table 4.3: **FTL Mapping Table Size.** *Mapping table size of page-level, hybrid, and nameless-writing devices with different file system images. The configuration in Table 4.2 is used.*

size, and its log block area size (for hybrid mapping). A nameless-writing device keeps a mapping table for the entire file system's virtual address space. Since we map all metadata to the virtual block space in our nameless-writing implementation, the mapping table size of the nameless-writing device is dependent on the metadata size of the file system image. We use Impressions [4] to create typical file system images of sizes up to 1 TB and calculate their metadata sizes.

Table 4.3 shows the mapping table sizes of the three FTLs with different file system images produced by Impressions. Unsurprisingly, the page-level mapping has the highest mapping table space cost. The hybrid mapping has a moderate space cost; however, its mapping table size is still quite large: over 100 MB for a 1-TB device. The nameless mapping table has the lowest space cost; even for a 1-TB device, its mapping table uses less than 3 MB of space for typical file systems, reducing both cost and power usage.

4.4.2 Application Performance

We now present the overall application performance of nameless-writing, page-level mapping and hybrid mapping FTLs with macro-benchmarks. We use varmail, fileserver, and webserver from the filebench suite [83].

Figure 4.2 shows the throughput of these benchmarks. We see that both page-level mapping and nameless-writing FTLs perform better than the hybrid mapping FTL with varmail and fileserver. These benchmarks contain 90.8% and 70.6% random writes, respectively. As we will see later in this section, the hybrid mapping FTL performs well with sequential writes and poorly with random writes. Thus, its throughput for these two benchmarks is worse than the other two FTLs. For webserver, all three FTLs deliver similar performance, since it contains only 3.8% random writes. We see a small overhead of the nameless-writing FTL as compared

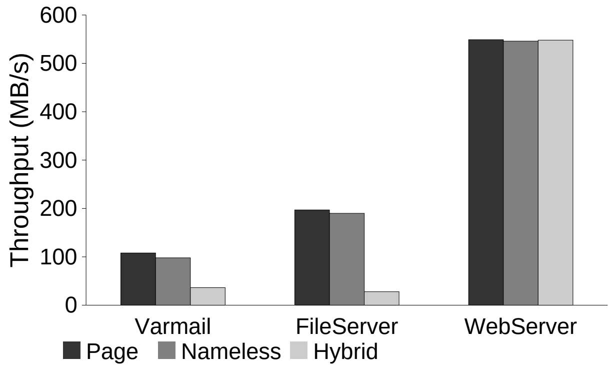


Figure 4.2: **Throughput of Filebench.** *Throughput of varmail, fileserver, and webmail macro-benchmarks with page-level, nameless-writing, and hybrid FTLs.*

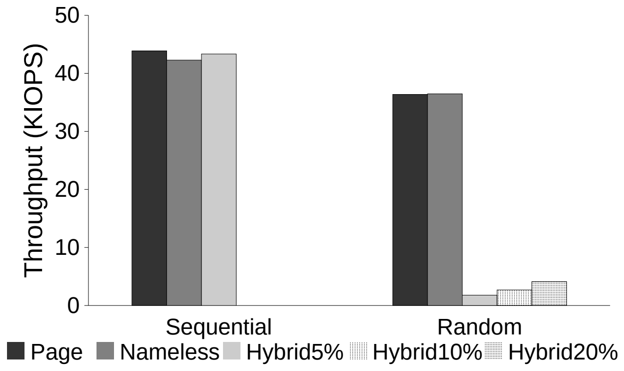


Figure 4.3: **Sequential and Random Write Throughput.** *Throughput of sequential writes and sustained 4-KB random writes. Random writes are performed over a 2-GB range.*

to the page-level mapping FTL with all benchmarks, which we will discuss in detail in Sections 4.4.5 and 4.4.6.

In summary, we demonstrate that the nameless-writing device achieves excellent performance, roughly on par with the costly page-level approach, which serves as an upper-bound on performance.

4.4.3 Basic Write Performance

Write performance of flash-based SSDs is known to be much worse than read performance, with random writes being the performance bottleneck. Nameless writes aim to improve write performance of such devices by giving the device more data-placement freedom. We evaluate the basic write performance of our emulated nameless-writing device in this section. Figure 4.3 shows the throughput of sequential writes and sustained 4-KB random writes with page-level mapping, hybrid mapping, and nameless-writing FTLs.

First, we find that the emulated hybrid-mapping device has a sequential throughput of 169 MB/s and a sustained 4-KB random write throughput of 2,830 IOPS. A widely used real middle-end SSD has sequential throughput of up to 70 MB/s and random throughput of up to 3,300 IOPS [44].

Second, the random write throughput of page-level mapping and nameless-writing FTLs is close to their sequential write throughput, because both FTLs allocate data in a log-structured fashion, making random writes behave like sequential writes. The overhead of random writes with these two FTLs comes from their garbage collection process. Since whole blocks can be erased when they are overwritten in sequential order, garbage collection has the lowest cost with sequential writes. By contrast, garbage collection of random data may incur the cost of live data migration.

Third, we notice that the random write throughput of the hybrid mapping FTL is significantly lower than that of the other FTLs and its own sequential write throughput. The poor random write performance of the hybrid mapping FTL results from the costly full-merge operation and its corresponding garbage collection process [40]. Full merges are required each time a log block is filled with random writes, thus a dominating cost for random writes.

One way to improve the random write performance of hybrid-mapped SSDs is to over-provision more log block space. To explore that, we vary the size of the log block area with the hybrid mapping FTL from 5% to 20% of the whole device and found that random write throughput gets higher as the size of the log block area increases. However, only the data block area reflects the effective size of the device, while the log block area is part of device over-provisioning. Therefore, hybrid-mapped SSDs often sacrifice device space cost for better random write performance. Moreover, the hybrid mapping table size increases with higher log block space, requiring larger on-device RAM. Nameless writes achieve significantly better random write performance with no additional over-provisioning or RAM space.

Finally, Figure 4.3 shows that the nameless-writing FTL has low overhead as compared to the page-level mapping FTL with sequential and random writes. We

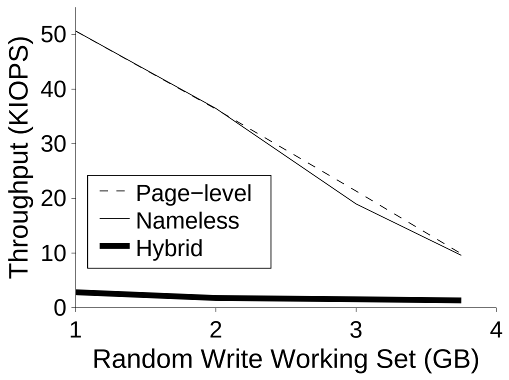


Figure 4.4: **Random Write Throughput.** *Throughput of sustained 4-KB random writes over different working set sizes with page-level, nameless, and hybrid FTLs.*

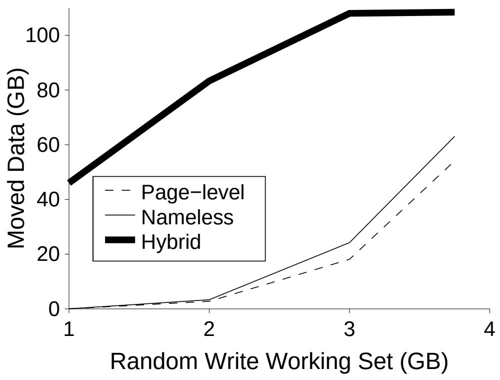


Figure 4.5: **Migrated Live Data.** *Amount of migrated live data during garbage collection of random writes with different working set sizes with page-level, nameless, and hybrid FTLs.*

explain this result in more detail in Section 4.4.5 and 4.4.6.

4.4.4 A Closer Look at Random Writes

A previous study [40] and our study in the last section show that random writes are the major performance bottleneck of flash-based devices. We now study two subtle yet fundamental questions: do nameless-writing devices perform well with

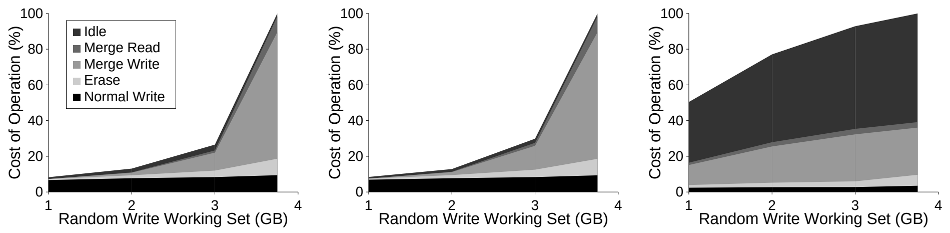


Figure 4.6: **Device Utilization.** *Break down of device utilization with the page-level, the nameless, and the hybrid FTLs under random writes of different ranges.*

different kinds of random-write workloads, and why do they outperform hybrid devices.

To answer the first question, we study the effect of working set size on random writes. We create files of different sizes and perform sustained 4-KB random writes in each file to model different working set sizes. Figure 4.4 shows the throughput of random writes over different file sizes with all three FTLs. We find that the working set size has a large effect on random write performance of nameless-writing and page-level mapping FTLs. The random write throughput of these FTLs drops as the working set size increases. When random writes are performed over a small working set, they will be overwritten in full when the device fills and garbage collection is triggered. In such cases, there is a higher chance of finding blocks that are filled with invalid data and can be erased with no need to rewrite live data, thus lowering the cost of garbage collection. In contrast, when random writes are performed over a large working set, garbage collection has a higher cost since blocks contain more live data, which must be rewritten before erasing a block.

To further understand the increasing cost of random writes as the working set increases, we plot the total amount of live data migrated during garbage collection (Figure 4.5) of random writes over different working set sizes with all three FTLs. This graph shows that as the working set size of random writes increases, more live data is migrated during garbage collection for these FTLs, resulting in a higher garbage collection cost and worse random write performance.

Comparing the page-level mapping FTL and the nameless-writing FTL, we find that nameless-writing has slightly higher overhead when the working set size is high. This overhead is due to the cost of in-place garbage collection when there is wasted space in the recycled block. We will study this overhead in details in the next section.

We now study the second question to further understand the cost of random

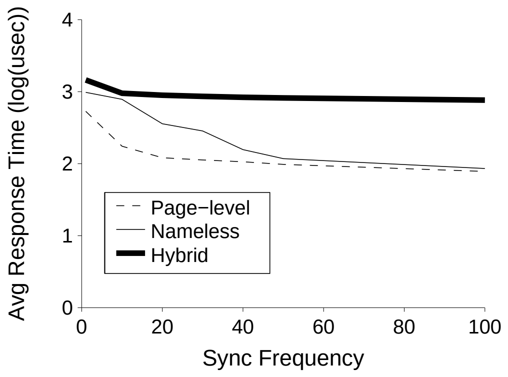


Figure 4.7: **Average Response Time of Synchronous Random Writes.** *4-KB random writes in a 2-GB file. Sync frequency represents the number of writes we issue before calling an fsync.*

writes with different FTLs. We break down the device utilization into regular writes, block erases, writes during merging, reads during merging, and device idle time. Figure 4.6 shows the stack plot of these costs over all three FTLs. For page-level mapping and nameless-writing FTLs, we see that the major cost comes from regular writes when random writes are performed over a small working set. When the working set increases, the cost of merge writes and erases increases and becomes the major cost. For the hybrid mapping FTL, the major cost of random writes comes from migrating live data and idle time during merging for all working set sizes. When the hybrid mapping FTL performs a full merge, it reads and writes pages from different planes, thus creating idle time on each plane.

In summary, we demonstrate that the random write throughput of the nameless-writing FTL is close to that of the page-level mapping FTL and is significantly better than the hybrid mapping FTL, mainly because of the costly merges the hybrid mapping FTL performs for random writes. We also found that both nameless-writing and page-level mapping FTLs achieve better random write throughput when the working set is relatively small because of a lower garbage collection cost.

4.4.5 In-place Garbage Collection Overhead

The performance overhead of a nameless-writing device may come from two different device responsibilities: garbage collection and wear leveling. We study the overhead of in-place garbage collection in this section and wear-leveling overhead

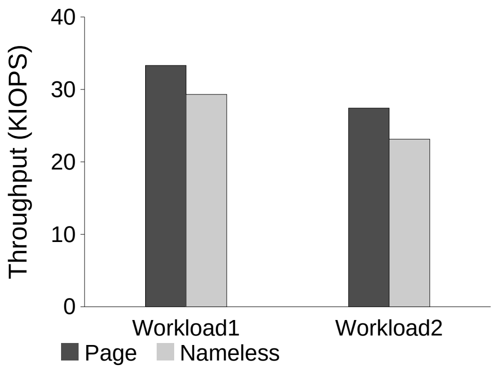


Figure 4.8: **Write Throughput with Wear Leveling.** *Throughput of biased sequential writes with wear leveling under page-level and nameless FTLs.*

	Metadata	RemapTbl
Workload1	2.02 MB	321 KB
Workload2	5.09 MB	322 KB

Table 4.4: **Wear Leveling Callback Overhead.** *Amount of additional metadata writes because of migration callbacks and maximal remapping table size during wear leveling with the nameless-writing FTL.*

in the next section.

Our implementation of the nameless-writing device uses an in-place merge to perform garbage collection. As explained in Section 4.2.2, when there are no waiting writes on the device, we may waste the space that has been recently garbage collected. We use synchronous random writes to study this overhead. We vary the frequency of calling *fsync* to control the amount of waiting writes on the device; when the sync frequency is high, there are fewer waiting writes on the device queue. Figure 4.7 shows the average response time of 4-KB random writes with different sync frequencies under page-level mapping, nameless-writing, and hybrid mapping FTLs. We find that when sync frequency is high, the nameless-writing device has a larger overhead compared to page-level mapping. This overhead is due to the lack of waiting writes on the device to fill garbage-collected space. However, we see that the average response time of the nameless-writing FTL is still lower than that of the hybrid mapping FTL, since response time is worse when the hybrid FTL performs full-merge with synchronous random writes.

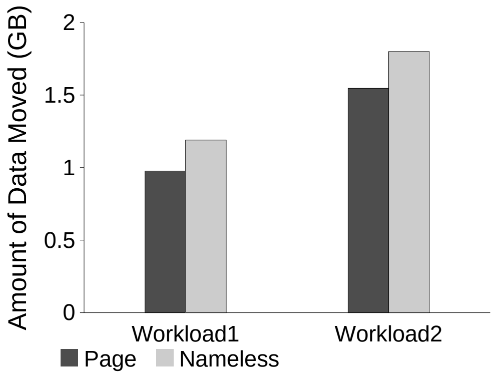


Figure 4.9: **Migrated Live Data during Wear Leveling.** *Amount of migrated live data during wear leveling under page-level and nameless FTLs.*

4.4.6 Wear-leveling Callback Overhead

Finally, we study the overhead of wear leveling in a nameless-writing device. To perform wear-leveling experiments, we reduce the lifetime of SSD blocks to 50 erase cycles. We set the threshold of triggering wear leveling to be 75% of the maximal block lifetime, and set blocks that are under 90% of the average block remaining lifetime to be candidates for wear leveling.

We create two workloads to model different data temperature and SSD wear: a workload that first writes 3.5-GB data in sequential order and then overwrites the first 500-MB area 40 times (Workload 1), and a workload that overwrites the first 1-GB area 40 times (Workload 2). Workload 2 has more hot data and triggers more wear leveling. We compare the throughput of these workloads with page-level mapping and nameless-writing FTLs in Figure 4.8. The throughput of Workload 2 is worse than that of Workload 1 because of its more frequent wear-leveling operation. Nonetheless, the performance of the nameless-writing FTL with both workloads has less than 9% overhead.

We then plot the amount of migrated live data during wear leveling with both FTLs in Figure 4.9. As expected, Workload 2 produces more wear-leveling migration traffic. Comparing page-level mapping to nameless-writing FTLs, we find that the nameless-writing FTL migrates more live data. When the nameless-writing FTL performs in-place garbage collection, it generates more migrated live data, as shown in Figure 4.5. Therefore, more erases are caused by garbage collection with the nameless-writing FTL, resulting in more wear-leveling invocation and more wear-leveling migration traffic.

Migrating live nameless data in a nameless-writing device creates callback traffic and additional metadata writes. Wear leveling in a nameless-writing device also adds a space overhead when it stores the remapping table for migrated data. We show the amount of additional metadata writes and the maximal size of the remapping table of a nameless-writing device in Figure 4.4. We find both overheads to be low with the nameless-writing device: an addition of less than 6 MB metadata writes and a space cost of less than 350 KB.

In summary, we find that both the garbage-collection and wear-leveling overheads caused by nameless writes are low. Since wear leveling is not a frequent operation and is often scheduled in system idle periods, we expect both performance and space overheads of a nameless-writing device to be even lower in real systems.

4.4.7 Reliability

To determine the correctness of our reliability solution, we inject crashes in the following points: 1) after writing a data block and its metadata block, 2) after writing a data block and before updating its metadata block, 3) after writing a data block and updating its metadata block but before committing the metadata block, and 4) after the device migrates a data block because of wear leveling and before the file system processes the migration callback. In all cases, we successfully recover the system to a consistent state that correctly reflects all written data blocks and their metadata.

Our results also show that the overhead of our crash recovery process is relatively small: from 0.4 to 6 seconds, depending on the amount of inconsistent metadata after crash. With more inconsistent metadata, the overhead of recovery is higher.

4.5 Summary

In this chapter, we introduced nameless writes, a new write interface built to reduce the inherent costs of indirection. With nameless writes, the file system does not perform allocation and sends only data and no logical address to the device. The device then sends back the physical address, which is stored in the file system metadata.

In implementing nameless writes, we met a few challanges, such as the recursive update problem and the device block migration problem. We solve these problems by introducing address space separation (logical and physical address space) and new types of interface (migration callback). Through the implementation of nameless writes on the Linux ext3 file system and an emulated nameless-writing device, we demonstrated how to port a file system to use nameless writes.

Through extensive evaluations, we found that nameless writes largely reduce the mapping table space cost as compare to the page-level mapping and the hybrid mapping FTLs. We also found that for random writes, nameless writes largely outperforms the hybrid mapping FTL and matchs the page-level mapping FTL. Overall, we show the great advantage of nameless writes from both worlds: the good performance like the page-level mapping FTL and the small mapping table space that is even less than the hybrid mapping FTL.

Chapter 5

Hardware Experience of Nameless Writes

As with most research work of flash memory, we evaluated our nameless writes design not with real hardware but with our own SSD emulator. Simulators and emulators are convenient and flexible to build and use. However, simulation and emulation have their limitations.

The nameless writes design makes substantial changes to the I/O interface, the SSD FTL, and the file system. Our SSD emulator lies directly below the file system and talks to it using software function calls, and thus simplifies the system which the nameless writes design is supposed to change. Because of the changes required by nameless writes at different storage layers, nameless writes are an ideal choice for studying the differences between real hardware systems and simulation/emulation, as well as the challenges in building a new storage interface for real storage systems.

Therefore, we decided to build a hardware prototype of nameless writes and use it to validate our nameless writes design. In this chapter, we discuss our hardware experience with implementing nameless writes with the OpenSSD Jasmine hardware platform [86], the challenges of building nameless writes with real hardware, and our solutions to them [79].

Our evaluation results of the nameless writes hardware prototype agrees with the conclusions we made in Chapter 4: nameless writes largely remove the excess indirection in SSDs and its space and performance costs.

The rest of this chapter is organized as follows. We first describe the architecture of the hardware board we use in Section 5.1. We then discuss the challenges of porting nameless writes to hardware in Section 5.2. We present our solutions to

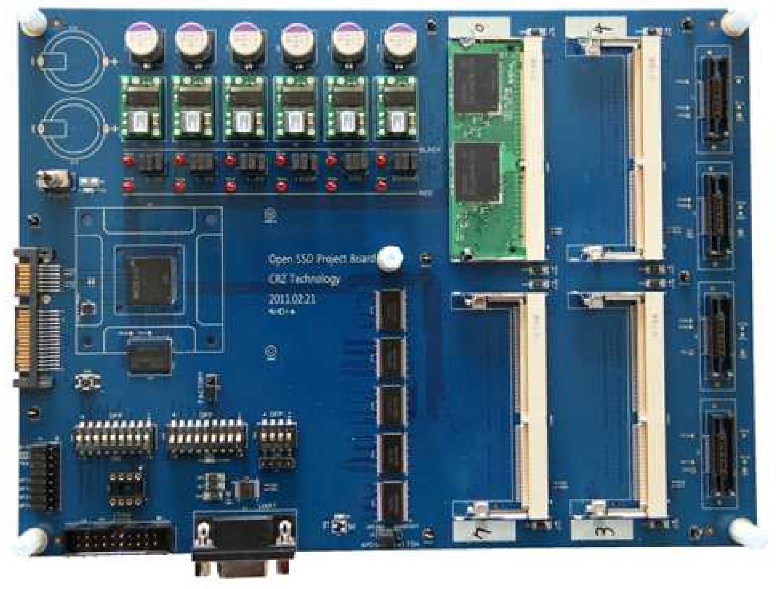


Figure 5.1: **OpenSSD Architecture** *The major components of OpenSSD platform are the Indilinx Barefoot SSD controller; internal SRAM, SDRAM, and NAND flash; specialized hardware for buffer management, flash control, and memory utility functions; and debugging UART/JTAG ports.*

these challenges and the implementation of the nameless writes hardware prototype in Section 5.3. In Section 5.4, we evaluate the nameless writes hardware prototype. Finally, we summarize this chapter in Section 5.5.

5.1 Hardware Platform

We use the OpenSSD platform [86] (Figure 5.1) 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 a nameless-writing SSD to verify its practicality and validate if it performs as we projected in emulation earlier.

5.1.1 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 [86]. The board is composed of commodity SSD parts: an Indilinx Barefoot ARM-based SATA controller, introduced in 2009 for second generation SSDs and

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 5.1: **OpenSSD device configuration.** *This table summarizes the hardware configuration in the OpenSSD platform.*

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 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 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 5.2: 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 as our baseline; it uses log

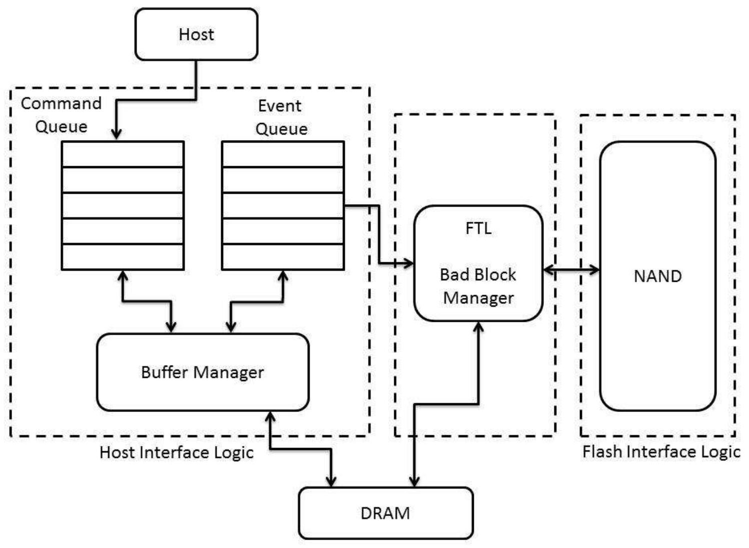


Figure 5.2: **OpenSSD Internals.** *Major components of OpenSSD internal design are host interface logic, flash interface logic, and flash translation layer.*

structured allocation and thus has good random write performance. It is similar to the page-level mapping FTL we used in our emulation in Chapter 4.

5.2 Challenges

Before delving into the implementation details of the nameless writes design with the OpenSSD platform and the SATA interface, we first discuss the challenges we encountered in integrating nameless writes with real hardware and a real hardware interface.

Nameless writes present unique implementation challenges because they change the interface between the host OS and the storage device by adding new commands, new command responses, and unrequested up-calls. Table 4.1 in Chapter 4 lists the nameless writes interfaces.

When moving from emulation (Chapter 3) to real hardware, not only the hardware SSD needs to be ported to nameless writes, but the hardware interface and the OS stack do as well. Figure 5.2 describes how our SSD emulator and the real SSD work with the OS.

The emulator sits at the OS block layer and interacts with the OS through software interfaces. I/Os are passed between the file system and the emulator using the *bio* structure. Adding new types of interfaces is easy. For example, the nameless

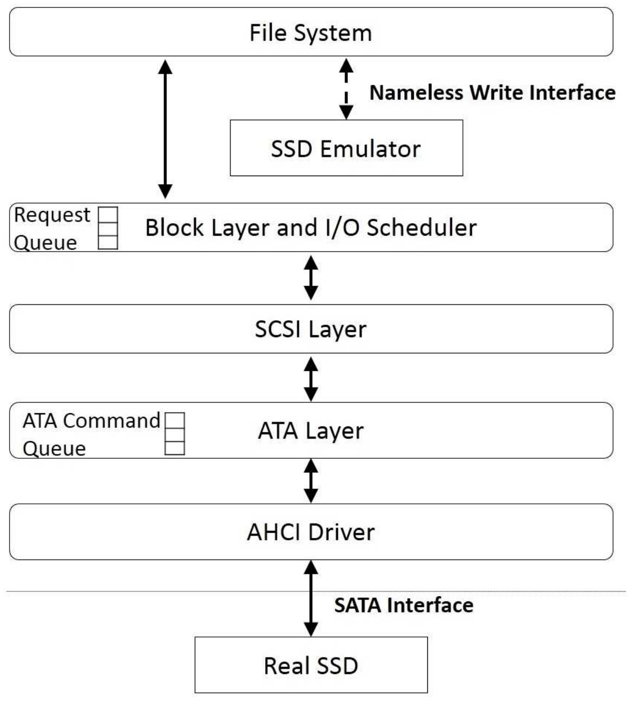


Figure 5.3: **Architecture of OS Stack with Emulated and Real SSD.** *This graph illustrates the OS stack above a real SSD with SATA interface. As opposed to the stack to the real device, the emulator is implemented directly below the file system.*

write command is implemented by adding a command type flag in the bio structure; the physical address returned by a nameless write is implemented by reusing the logical block address field of the bio structure, which is then interpreted specially by a file system that works with nameless writes. Adding the migration callback is also relatively easy: the device emulator calls a kernel function, which then uses a kernel work queue to process the callback requests.

The interaction between the OS and the real hardware device is much more involved than with the emulator. I/O requests enter the storage stack from the file system and go through a scheduler and then the SCSI and ATA layers before the AHCI driver finally submits them to the device. To implement our hardware prototype, we have to integrate the nameless writes interface into this existing storage architecture. Implementing a new interface implies the need to change the file system and the OS stack, the ATA interface, and the hardware SSD. The biggest challenge in this process is that some part of the storage stack is fixed in the hardware and cannot be accessed or changed. For example, most of the ATA interface is implemented in the hardware ports and cannot be changed (see Chapter 2 for more details). Certain parts of the OpenSSD platform cannot be changed or accessed either, such as the OOB area.

5.2.1 Major Problems

We identified four major problems while implementing nameless writes with the real hardware system: how to get new commands through the OS storage stack into the device, how to get new responses back from the device, how to have upcalls from the device into the OS, 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 5.2, which interpret and act on each command differently. For example, the I/O scheduler can merge requests to adjacent blocks. If it is not aware that the *virtual-write* and *nameless-write* 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. The nameless writes interface returns the physical address for data following a nameless write. However, the SATA write command normal response has no fields in which an address can be returned. While, the error response allows an address to be returned, both the AHCI driver and the ATA layer 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 [39, 76].

Third, there are no ATA commands that are initiated by the device. Nameless writes require upcalls from the device to the OS for migration callbacks. Implementing upcalls is challenging, since all ATA commands are initiated by the OS to the device.

Finally, the OpenSSD platform provides hardware support for the SATA protocol (see Figure 5.2) 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.

5.3 Implementation Experiences

In this section, we discuss how we solve the challenges of transferring the nameless writes design from emulation to real hardware and our experience with implementing nameless writes on the OpenSSD hardware with the Linux kernel and the SATA interface. We focus our discussion on changes in the layers below the file system, since the file system changes are the same for the emulator and the real hardware.

5.3.1 Adding New Command Types

To add a new command type, we change the OS stack, the ATA interface, and the OpenSSD firmware. We now discuss the techniques we used for introducing new command types. We also discuss the implementation of new nameless writes commands that do not involve return field changes or upcalls, both of which we leave for later sections in this chapter.

Forward commands through the OS: At the block-interface layer, we seek to leave as much code unmodified as possible. Thus, we augment block requests with an additional command field, effectively adding our new commands as sub-types of existing commands. The nameless write, the nameless overwrite, and the virtual write commands are encoded using three special sub-types with the normal write command. We also use the normal write command to encode the trim command. The trim command, however, does not send any data but only block addresses to the device. The virtual and physical read commands are encoded in a similar way with normal read commands.

We modified the I/O scheduler to only merge requests with the same command and sub-type. The SCSI and ATA layers then blindly pass the sub-type field down to the next layer. We also modified the AHCI driver to communicate commands to the OpenSSD device. As with higher levels, we use the approach of adding a sub-type to existing commands.

ATA interface: 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 reads and writes, and allows multiple variations of the commands.

5.3.2 Adding New Command Return Field

Nameless writes pass data but no address, and expect the device to return a physical address or an error indicating that 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; all other fields are reserved and can not be used to send physical addresses through the ATA interface.

Our first attempt was to alter the error return of an ATA write to return physical addresses for nameless writes. 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 attempt was to re-purpose an existing SATA command that *already* returns a 64-bit address. Only one command in the ATA protocol, *READ_NATIVE_MAX_ADDR*, returns an address. The OS would first send *READ_NATIVE_MAX_ADDR*, to which the nameless-writing 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.

Therefore, neither of these two attempts to integrate a physical address in the write return path with the ATA interface succeeded. We defer the discussion of our final solution to Section 5.3.4.

5.3.3 Adding Upcalls

The *migration-callback* command raised additional problems. Unlike *all* existing calls in the SATA interface, a nameless-writing 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.

To implement migration callbacks, we first considered piggybacking the upcalls 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 method is too costly in performance given the expected rarity of up-calls. We discuss our final solution in the next section.

5.3.4 Split-FTL Solution

Based on the complexity of implementing the full nameless writes 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 [33], 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-writing FTL at the block layer below the file system and the I/O scheduler and above the SCSI, ATA, and AHCI layers. Figure 5.4 shows the design. The FTL in the host OS implements the full set of interfaces listed in Table 4.1 in Chapter 4; the device implements 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.

We built the nameless-write FTL below the I/O scheduler, since placing the FTL above the I/O scheduler creates problems when allocating physical addresses. If the FTL performs its address allocation before requests go into the I/O scheduler, the I/O scheduler may re-order or merge requests. If the FTL assigns multiple physical addresses from one virtual flash page, the scheduler may not coalesce the writes into a single page-sized write. The device may see multiple writes (with different physical addresses) to the same virtual flash page without erases and thus result in data corruption.

The nameless-writing 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

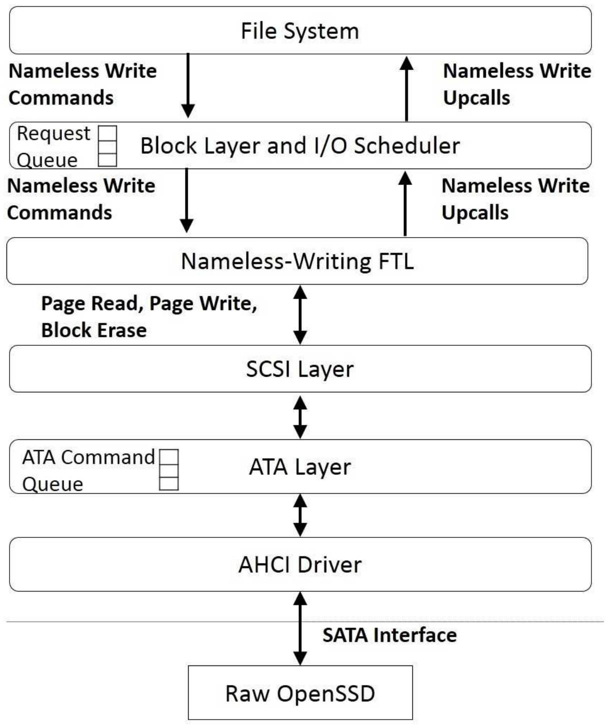


Figure 5.4: **Nameless Writes Split-FTL Architecture.** *This figure depicts the architecture of the split-FTL design of nameless writes. Most of the nameless writes functionality is implemented as a layer within the host operating system (below the file system and the block layer). The nameless writes interfaces are implemented between the nameless-writing FTL and the block layer. The device (OpenSSD) operates as a raw flash device.*

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, the free block list, block erase counts, and the 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 reads 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.

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 [72] or use new communication channels based on RPC-like mechanisms [65].

5.3.5 Lessons Learned

The implementation of nameless writes with OpenSSD and the ATA interfaces imparted several valuable lessons.

- 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.

- The device response paths in the OS are difficult to change. Therefore, designs that radically extend existing communication from the device should consider the data that will be communicated.

- Upcalls from the device to the OS do not fit the existing communication channels between the host and the device, and changing the control path for returning values is significantly more difficult than introducing new forward commands. Thus, it may be worthwhile to consider new reverse communication channels based on RPC-like mechanisms [65] to implement block interface for PCI-e devices following the NVM-express specification [72].

- Building the nameless-writing FTL below the block layer is simpler than at the device firmware since the block layer has simpler interfaces and interacts with the file system directly.

- Allowing the host OS to write directly to physical addresses is dangerous, because it cannot guarantee correctness properties such as ensuring the erasure of a flash page before it is written. This is particularly dangerous if the internal write granularity is different than the granularity used by the OS.

- With the knowledge of SSD hardware configuration, the kernel I/O scheduler can be changed to improve I/O performance with an in-kernel FTL.

5.4 Evaluation

Our overall goal of implementing nameless writes in a hardware prototype is to validate our design choices. We evaluate the memory space and performance of nameless writes prototype with the split-FTL design.

We evaluate the nameless writes prototype to validate the performance claims of the interface and memory consumption as projected earlier in Section 4.4. We compare the nameless writes prototype against the OpenSSD baseline 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 5.5 shows the random (4 KB blocks) and sequential write and read performance with the baseline OpenSSD FTL and nameless writes. For sequential writes, sequential reads, and random reads, the nameless-writing 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 either 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 5.2 presents the memory usage of the page-mapped FTL and the nameless-writing FTL with different file system image sizes (from 4 GB to 48 GB), which

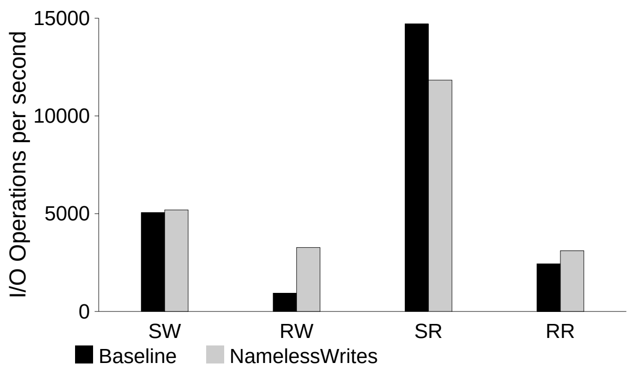


Figure 5.5: **Read and Write Performance.** *This figure presents the IOPS of the OpenSSD FTL and the nameless writes split-FTL for fio benchmarks (sequential and random reads and writes with a 4 KB request size).*

	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 5.2: **FTL Memory Consumption.** *This table presents the memory usage of the baseline page-level mapping FTL and the nameless-writing FTL. We use Impressions to generate typical file system images with different sizes.*

we created using Impressions [4]. The memory consumption includes the address mapping tables and all additional FTL metadata. The nameless-writing FTL uses much less memory than the page-mapped FTL. Unlike the page-mapped FTL, the nameless-writing 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 writes design performs similarly to the page-mapped FTL and provides significant memory savings as projected earlier in Chapter 4.

5.5 Summary

In this chapter, we described our experience with building nameless writes with the OpenSSD hardware board, the challenges in moving nameless writes to hardware, and our solutions to them.

The biggest challenges we met in our hardware experience are changing the I/O return path and adding upcalls from the device, neither of which we foresaw when we built nameless writes with emulation.

Because of the restrictions of ATA interface, we change the nameless writes design to use two split parts, an FTL at the OS block layer that manages most of the responsibilities of a nameless-writing device, and a simple raw FTL that manages a raw SSD device. We show through evaluation that this design works well with a real hardware system, and achieves the same benefits of de-indirection as with our original nameless writes design using emulation.

Overall, we found our hardware experience to be rewarding; we learned a set of new lessons in how hardware and real systems can be different from simulation and emulation. Even because of a single restriction in real hardware, the whole system may need to be re-designed.

Chapter 6

A File System De-Virtualizer

We demonstrated in Chapters 4 and 5 that nameless writes are able to largely remove SSD indirection and its space and performance costs. However, the nameless writes solution has a few drawbacks. First, nameless writes require fundamental changes to the file system, the OS, the device, and the device interface. Second, the nameless-write approach creates unnecessary overhead because it performs de-indirection for all data writes; instead, de-indirection can be performed at device idle time to hide its overhead.

To overcome the drawbacks of nameless writes, we propose the *File System De-Virtualizer* (*FSDV*), a mechanism to dynamically remove the indirection in flash-based SSDs with small changes to existing systems. FSDV is a user-level tool that walks through file system structures and changes file system pointers to physical addresses; when pointers are *de-virtualized*, the logical to physical address mappings in the SSD can be removed. FSDV can be invoked periodically, when the memory pressure in SSD is high, or when the device is idle.

We implemented a prototype of FSDV and modified the ext3 file system and our emulated flash-based SSD for it. Our evaluation results of FSDV demonstrate that FSDV largely reduces device mapping table space in a dynamic way. It achieves this goal with small performance overhead on foreground I/Os.

The rest of this chapter is organized as follows. We present the basic design of FSDV in Section 6.1. We then describe our implementation of the FSDV tool, the changes to the ext3 file system, and our SSD emulator for FSDV in Section 6.2. In Section 6.3, we present our evaluation results of FSDV. Finally, we summarize this chapter in Section 6.4.

6.1 System Design

In this section, we present the overall design of FSDV, and how it interacts with the device and the file system that support FSDV. We design FSDV with the following goals in mind.

1. Indirection mappings can be (largely) removed in a dynamic way. Doing so allows FSDV to be able to remove the device virtualization maintained either inside the device hardware or in the host software.

2. The performance overhead of FSDV should be low, so that there will be negligible impact on normal I/Os.

3. There should only be small changes in file systems, OSes, device firmware, and device I/O interfaces. Doing so will allow for an easy integration of FSDV into existing systems.

FSDV is a user-level tool that runs periodically or when needed to remove the excess virtualization of a virtualized storage device. When FSDV is not running, a normal file system runs with the storage device in a largely unmodified way. The file system performs block allocation in the *logical address space*. The device allocates *device addresses* and maintains an indirection mapping from logical to device addresses. When the mapping table space pressure is high, FSDV can be invoked to perform de-virtualization to remove indirection mappings. FSDV can also be invoked periodically or when the device is idle.

6.1.1 New Address Space

FSDV de-virtualizes a block by changing the file system pointer that points to it (*i.e.*, the metadata) to use the device address. After FSDV de-virtualizes a block, it is moved from the logical address space to the *physical address* space and directly represents a device address (*i.e.*, no mapping is maintained for the block). As workloads perform new I/Os, the file system allocates new data in the logical address space and overwrites existing data in the physical address space. For the former, the device adds a mapping from the logical address to the device address. For the latter, the device adds a mapping from the old physical address to the new device address.

Figure 6.1 gives an example of FSDV address spaces and indirection mappings. Since a block can be in different address spaces, we need a method to distinguish logical addresses from physical ones. We discuss our method in Section 6.2.

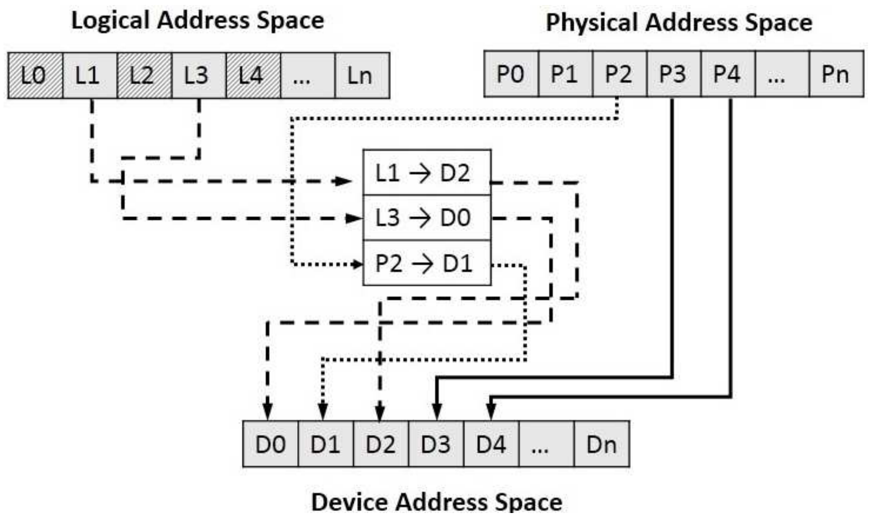


Figure 6.1: **FSDV Address Spaces.** *Device address space represents the actual physical addresses in the device. The file system sees both logical and physical addresses. In this example, the logical addresses L1 and L3 are mapped to the device addresses D2 and D0 through the device mapping table; the physical address P2 is mapped to the device address D1. The physical addresses P3 and P4 are unmapped and directly represent the device addresses D3 and D4. The logical addresses L0, L2, and L4 (shadowed) represent the logical addresses that have been freed (either by a file system delete or a FSDV de-virtualization).*

For future reads after FSDV runs, the device checks if there is a mapping entry for the block. If there is, the device serves the reads after remapping, and if not, the device reads directly from the device address.

Another cause for address mapping change and addition is the physical address migration during different flash device operations. As a flash device is accessed by different types of data, its flash blocks will be in different states and the device performs garbage collection or wear leveling operations. Both these operations involve migration of physical blocks. When a directly mapped block is migrated to a new device address, a new mapping will be added to map from its old device address to its current device address. FSDV also needs to remove these new mappings.

The challenge in handling address mapping addition caused by the device is that there is no way of knowing what files the migrated blocks (and their mapping) belong to. A simple way to handle these mappings is to scan and perform de-virtualization to the whole file system; these mappings will eventually be removed in the process. However, the performance cost of whole-file scanning can be high, especially for large file systems. We choose another method to solve the problem, involving associating each block to the file to which it belongs. Specifically, we change the file system write interface to also include the inode number and let the device store it with the block. When the device migrates a block and adds a new mapping, it records the inode number the block belongs to; FSDV will process these files in its later run.

6.1.2 FSDV Modes

FSDV can run offline (with unmounted file system) or online (with mounted file system). The offline FSDV works with an unmounted file system. After the file system has been unmounted (either by user or forcefully by FSDV), FSDV goes through the file system data structures and processes file system metadata to de-virtualize file system pointers. The de-virtualized file system is then mounted.

The offline FSDV provides a simple and clean way to perform file system de-virtualization. However, it requires file systems to be unmounted before it can start processing and is not suitable for most storage systems. Thus, we also design an online version of FSDV which runs while the file system is mounted and foreground I/Os run in a (largely) unaffected fashion. The major difference between the online FSDV and the offline one is that it needs to make sure that it does not leave any inconsistency in page cache or for ongoing I/Os. FSDV interacts with the file system through the FSDV-supporting device to isolate the blocks it processes from the file system.

6.2 Implementation

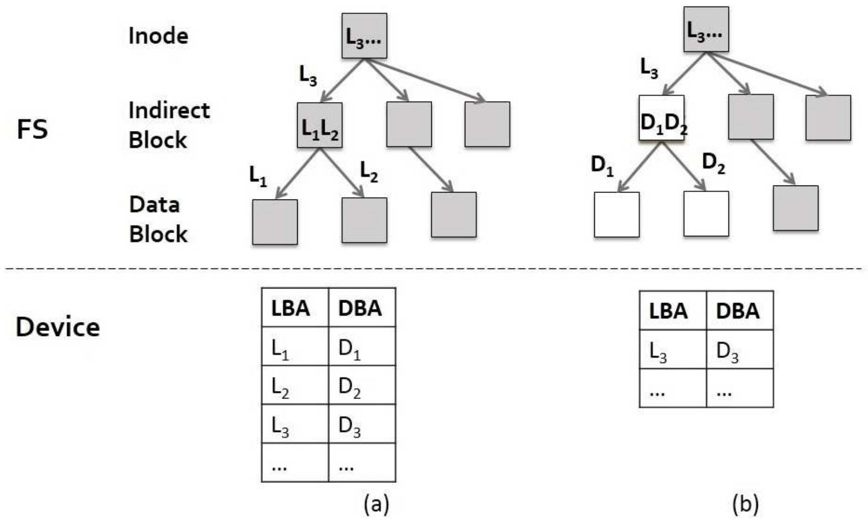
In this section, we discuss our implementation of the FSDV tool, the offline and the online version of it. To support FSDV, changes in device firmwares, file systems, and OSes are needed. We also discuss these changes in this section.

6.2.1 File System De-virtualizer

We now describe our implementation of the offline and the online FSDV and a few optimizations we make for better FSDV performance.

Offline File System De-Virtualizer

The offline FSDV works with unmounted file systems. As explained in Section 2.2.1, for most file systems like ext2, ext3, and ext4, a file can be viewed as a tree structure with the inode of the file at the tree root, indirect blocks (or extent blocks) in the



LBA	DBA
L_1	D_1
L_2	D_2
L_3	D_3
...	...

LBA	DBA
L_3	D_3
...	...

Figure 6.2: **FSDV Processing a File Tree.** *The left part of the graph (a) represents the status of a file in the file system and the device mapping table before FSDV runs. The right part (b) represents a status in the middle of a FSDV run. L1 and L2 have been devirtualized to D1 and D2. The indirect block containing these pointers has also been rewritten. The mappings from L1 to D1 and L2 to D2 in the device have been removed as well.*

middle of the tree, and data blocks at the leaf level. FSDV de-virtualizes a file by walking through the file tree structure and processing metadata blocks from bottom up (*i.e.*, from the metadata blocks that directly point to the data blocks to the inode). We choose to use the bottom-up fashion because in this way by the time when FSDV processes an upper-level metadata block, all its children have already been processed; FSDV can update this metadata block with the final device addresses of all its children.

For each pointer in a metadata block, FSDV sends the address that the pointer uses (either logical or physical address) to the device and queries for its current device address. If the device returns a device address, then the metadata block will be updated to use this address. After all the pointers in the metadata block have been processed, FSDV writes the metadata block back to the device (if it is changed) and informs the device to remove the corresponding mappings. When a mapping from a logical address is removed, the file system bitmap is updated to unset the corresponding bit. Figure 6.2 gives an example of FSDV processing a file tree.

To de-virtualize inodes, we change the way of locating an inode from using the inode number to using the device address of the inode block and the offset of the inode within the inode block. After FSDV de-virtualizes all per-file metadata blocks as described above, FSDV starts to process inodes and the metadata blocks pointing to them (*i.e.*, directory blocks). FSDV changes the pointer pointing to an inode to use the device address of the inode block and the inode's offset within it. If the inode is de-virtualized from an inode number (its original form), the file system inode bitmap will also be updated. Currently, we do not de-virtualize directory inodes, since processing directory data structures is more complicated and directories only account for a small part of typical file systems [5]; we leave de-virtualizing directory inodes for future work.

Finally, we do not need to deal with any metadata in file system journals. When unmounted, a file system's journal is checkpointed. Thus, there are no outstanding transactions and the journal is empty. We do not de-virtualize block group bitmap blocks, group description blocks, or superblocks either, since they only account for a small space in the file system.

Online De-Virtualizer

The online FSDV runs while the file system is mounted. Most of its mechanisms are the same as the offline FSDV. However, since we allow foreground I/Os to be performed while the online FSDV is running, we need to make sure that such a situation does not leave the file system inconsistent. To achieve this goal, FSDV informs the file system (through the device) about the blocks it wants to isolate; the file system then flushes all the page caches corresponding to these blocks. When FSDV is processing these blocks, the device will prevent ongoing I/Os to them simply by stalling the I/Os until FSDV finishes its processing. The FSDV process registers a special process ID with the device, so that I/Os issued by FSDV will never be blocked.

We have two options in terms of blocking granularity for the online FSDV: at each file and at each metadata block. If we choose blocking at the file granularity, the device sends the inode identity to the file system, which then flushes all page caches belonging to this file. The device keeps track of the inode that FSDV is processing and stalls all file system I/Os with this inode until FSDV finishes processing it. The per-file method is conceptually simple and fits well with the way FSDV performs de-virtualization: one file at a time. However, it creates a higher performance overhead, especially for big files, since all blocks belonging to a file are flushed from the page cache and all I/Os of the file are stalled when FSDV is processing the file.

If we choose blocking at the metadata block level, FSDV sends all the block numbers which the metadata block points to and the metadata block number itself to the file system (again, through the device). The file system then flushes the corresponding blocks from the page cache. The device keeps track of these block addresses and prevent I/Os to them until FSDV finishes its processing of the metadata block.

Optimization Policies

We introduce a couple of optimizations to reduce the overhead of FSDV. First, since FSDV runs periodically, it does not need to process the files that have not been changed (overwritten or appended) since the last run of FSDV. We change the file system to record the updated files (particularly, inodes) and to send the list of such inodes to the device during journal commit. Notice that we do not need to worry about the consistency of such updated inode list; even if they are wrong, the file system will still be consistent, since FSDV will just process unnecessary (*i.e.*, unchanged) files.

To further reduce the run time of FSDV, we can choose not to process hot data with FSDV, since they will soon be overwritten after FSDV de-virtualizes them. The file system sends the update time of the inodes together with the changed inode list to the device. FSDV uses a *hot inode threshold* to only process files that are not accessed recently. For example, if we set the hot inode threshold to be 1/10 of the time window between two FSDV runs, the latter run will ignore the inodes that are updated within 1/10 of such time window.

6.2.2 Device Support

We have changed our SSD emulator (described in Chapter 3) to support FSDV. Most part of the emulated SSD and its FTL are not changed. The device still performs device address allocation for writes. We choose to use log-structured allocation and page-level mapping for better performance. For reads, the device looks up its mapping table and either read it directly from the device address or read the device address after mapping. For a write, the device records the inode number associated with the write in the OOB area adjacent to the data page that the device assigns the write to. When the device migrates a data page during a garbage collection or wear leveling operation, it also moves the inode number to the new OOB area. If a mapping is added because of this migration, the device also records the inode number for FSDV to process in its next round.

FSDV interacts with the FSDV-supporting device using normal I/O operations and simple ioctl commands. Table 6.1 summarizes the interfaces between FSDV and the device. When FSDV queries the device for the device address of a block, the device looks up its mapping and returns the mapped address or the no-mapping-found state to FSDV. After processing and writing new metadata block, FSDV tells the device to remove corresponding mappings.

For performance optimization of FSDV, the device also records the files (their inode identities) that have been updated from the last run of FSDV (from the new inode list that the file system sends to the device). FSDV reads and processes these new files. When FSDV finishes all its processing, the device deletes all the recorded new inodes. We choose to only store the new file record in device RAM and not permanently on flash memory, since even if the new file record is lost or is wrong, it will not affect the consistency or correctness of the file system (but FSDV may perform de-virtualization to unnecessary files).

The device works with the file system and FSDV for I/O flushing and blocking to support online FSDV. Specifically, when FSDV informs the device about its intention to process a file or a block, the device sends such information to the file system. Once the file system finishes the flushing and FSDV starts to process the file or the block, the device blocks foreground I/Os to the file or the block until FSDV finishes its processing.

Finally, the device also works with FSDV for reliability problems, (*e.g.*, it keeps certain FSDV operation logs and sends the replay information to FSDV during recovery). We defer the reliability discussion to Section 6.2.4.

6.2.3 File System Support

We ported ext3 to support FSDV. We now describe the changes we make to ext3 and the design choices that we make.

For I/O operations other than writes, there is no change needed with the file system and the OS, which is one of our major goals with FSDV. The file system performs its own allocation and maintains its logical address space. The file system data structures are mostly unchanged (the only exception being the inode identification method as described earlier in Section 6.2.1).

Get Device Address	
FSDV to device:	logical/physical address
device to FSDV	device address
description:	look device to FSDV device address
Remove Mapping	
FSDV to device:	logical/physical address
device to FSDV	if success
description:	remove mapping entry
Get Map Table Size	
FSDV to device:	null
device to FSDV	mapping table size
description:	get device mapping table size
Get New Inodes	
FSDV to device:	null
device to FSDV	new inode list
description:	get new inode list from device
Log Operation	
FSDV to device:	logical/physical address
device to FSDV	if success
description:	log addresses for FSDV reliability
Flush File/Block	
FSDV to device:	inode/block number
device to FSDV	null
description:	inform FS to flush the file/block
Check File/Block	
FSDV to device:	inode/block number
device to FSDV	null
description:	wait for file/block to be flushed
Block File/Block	
FSDV to device:	inode/block number
device to FSDV	null
description:	start blocking I/Os to the file/block
Unblock File/Block	
FSDV to device:	inode/block number
device to FSDV	null
description:	unblock I/Os to the file/block

Table 6.1: **Interfaces between FSDV and the FSDV-supporting devices** *The table presents the interface between FSDV and the FSDV-supporting device. All the commands are initiated by FSDV to the device.*

Write	
down:	data, length, inode number
up:	status, data
description:	write with associated inode number
Trim Block	
down:	block number
up:	null
description:	invalidate a block
Add New Inodes	
down:	new inode list
up:	null
description:	add new inodes to device
Flush File/Block	
down:	null
up:	inode/block number
description:	inform FS to flush the file/block
Done Flush File/Block	
down:	inode/block number
up:	null
description:	inform the device that file/block is flushed

Table 6.2: **Interface between the File System and the FSDV-supporting Device** *The table presents the interface between the file system that is ported to FSDV and the FSDV-supporting device. The last three commands are for the online FSDV only. The "Add New Inodes" command is for FSDV performance optimization.*

We make a few changes of ext3 to support FSDV. First, to distinguish logical addresses from physical ones, we add to each physical address the value of the total device size; thus, the logical and the physical address spaces never overlap. We also change the device size boundary check to accommodate physical addresses (the total size is doubled since we have two non-overlapping address spaces).

Second, we change the way the file system tracks address spaces and performs de-allocation to support FSDV. The file system uses the logical address bitmaps to track allocated logical addresses and uses a free space counter to track the total amount of actual allocated (device) addresses. When a block is devirtualized from its logical address, its corresponding bit in the file system bitmap is unset. Doing so will create more free logical addresses than the actual free space in the device. The file system uses the free block counter to keep track of the amount of free space and does not change it during FSDV operations. During allocation, the file system checks this counter to determine the actual free space left on the device. When the file system deletes a block in the physical address space, the file system updates the free block counter but does not change any bitmaps. When the file system deletes a block in the logical address space, it updates both the bitmap and the free block counter. The file system also sends a trim command to the device to inform it about the de-allocation.

Third, the file system tracks the inode that a block belongs to (when it is allocated) and sends the inode identity to the device during a write.

Finally, the file system works with the device to support online FSDV. Specifically, when the device sends the request of flushing a file or a block, the file system adds such information (inode identity or block number) to a work queue. A work queue handler then processes these files or blocks. For a file, the file system flushes all the blocks belonging to this file from the page cache and also clears the inode cache. For a block, the file system simply flushes it from the page cache. After the file system finishes the flushing process, it informs the device with the inode identity or the block numbers. Table 6.2 summarizes the interfaces between the file system and the device (those that are changed because of FSDV).

6.2.4 Reliability Issues

Finally, several reliability and consistency issues can happen during the de-virtualization process of FSDV. For example, the FSDV tool can crash before it completes its de-virtualization operations of a file, leaving the metadata of the file inconsistent. Another situation can happen with the online FSDV, where the FSDV tool dies and the device continues blocking file system I/Os.

We solve the reliability-related problems using several techniques. First, we make sure that the device never deletes the old metadata block until the new version of it has been committed. When the new metadata block is written, its old version is invalidated at the same time; this operation is an overwrite and most normal devices already invalidates old blocks atomically with overwrites. Second, FSDV logs all the old addresses a metadata block points to before FSDV processes the metadata block. Doing so makes sure that if FSDV crashes after writing the new metadata but before the device removes the old mappings of the pointers in this metadata block, the device can remove these mappings on recovery. When FSDV finishes its processing, the log on the device is removed. If a crash happens before FSDV finishes, the logs will be replayed during recovery. Finally, we set a timeout of device blocking file system I/Os to prevent dead or unresponsive FSDV tool.

6.3 Evaluation

In this section, we present our experimental evaluation of FSDV. Specifically, we answer the following questions.

1. What are the changes to the file systems, the OS, the device firmware, and the device I/O interface? Are the changes small and can they be easily implemented with existing systems?
2. How much indirection mapping space can FSDV remove? Can it be removed in a dynamic way?
3. How does the amount of inodes processed by FSDV affect the mapping table space reduction and performance of FSDV?
4. What is the difference between different FSDV modes? How does the offline mode compare to the online mode of FSDV? How do the online per-file and per-block modes compare?
5. What is the performance overhead of FSDV? How much does FSDV affect normal foreground I/Os?
6. How does the optimization techniques affect the performance and mapping table results?

We implemented the FSDV prototype as a user-level FSDV tool, and changed our emulated SSD device (described in Chapter 3), the ext3 file system, and the OS to support FSDV. The FSDV tool is implemented using the *fsck* code base. We change several aspects of our SSD emulator to support FSDV. We also make small changes to the ext3 file system and the OS (the block layer in particular) to support FSDV.

The total lines of code in the file system and the OS is 201 and is 423 in the device. Most of these changes are for handling de-allocation and FSDV performance optimization.

Experimental environment: All experiments were conducted on a 64-bit Linux 2.6.33 server, which uses a 3.3 GHz Intel i5-2500K processor and 16 GB of RAM. The emulated SSD used in our experiments has 5 GB size, 10 parallel flash planes, 4 KB flash pages, and 256 KB erase blocks. Flash page read and write operations take 25 and 200 microseconds and erase operation takes 1.5 milliseconds.

Workloads: We use a set of different types of workloads for our evaluation. To mimic typical file system images, we use the Impressions tool [4]. For more

Workloads	Total Size	Files	FileSize
F1	512 MB	1000	512 KB
F2	1 GB	2000	512 KB
F3	2 GB	2000	1 MB
F4	4 GB	2000	2 MB
F5	4 GB	4000	1 MB
I1	3.6 GB	3000	1.2 MB

Table 6.3: **Workloads Description** *This table describes the workloads property: the number of files and directories in the workloads and the average file size. Workloads F1 to F5 represent different FileServer workloads from the FileBench suite [83] (with varying number of files and directories). The workload I1 represents the file system image generated using Impressions [4].*

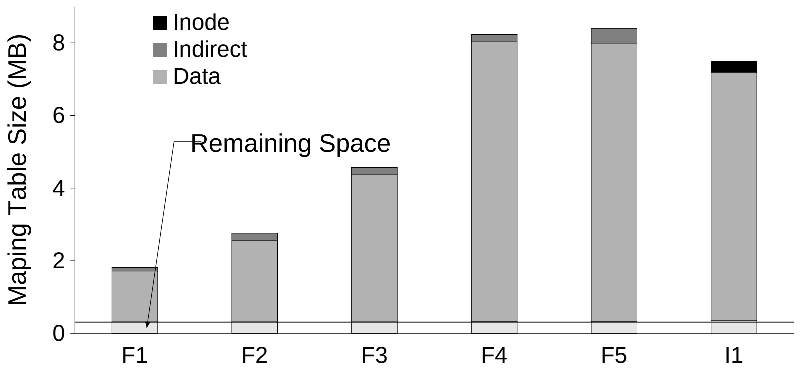


Figure 6.3: **Mapping Table Space Reduction.** *We invoke the offline FSDV after running different FileServer and Impressions workloads. The figure shows the mapping table space (in different types) reduction because of FSDV. The part below the horizontal line represents the remaining amount of the mapping table space that FSDV does not remove.*

controlled workloads, we use the FileServer macro-benchmark in the FileBench suite [83] with different numbers of directories and different average file sizes. Table 6.3 summarizes the settings used with these workloads.

6.3.1 Mapping Table Reduction and FSDV Run Time

We first evaluate the mapping table space reduction of FSDV; the major goal of FSDV is to reduce the mapping table space needed in a virtualized device. Fig-

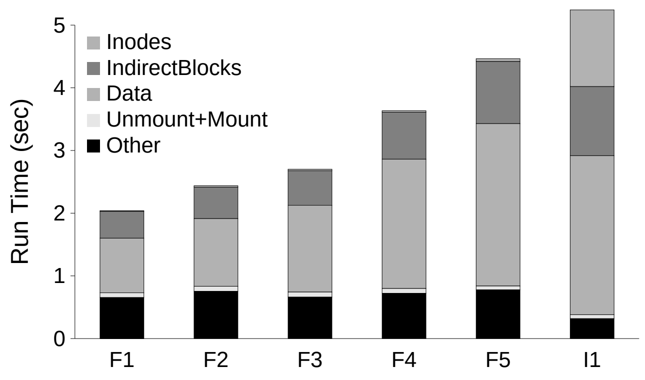


Figure 6.4: **FSDV Run Time.** *This figure plots the run time of the offline FSDV when de-virtualizing different FileServer and Impressions workloads. We break down the run time into time spent on de-virtualizing inodes, indirect blocks, and data blocks, time to unmount and mount the file system, and the rest of the FSDV time.*

ure 6.3 presents the amount of removed mapping tables with different FileServer workloads and the Impressions file system image. Specifically, we show the amount of removed mapping entries for data blocks, indirect blocks, inode blocks, and the amount of remaining mappings.

We find that FSDV reduces device mapping table size by 75% to 96% (*e.g.,* from 8.4 MB to 0.3 MB for the F5 workload). Most of the mapping table reduction is with data blocks, which conforms with the fact that typical file system images consist of data blocks [5]. We also find that larger files result in bigger data block and indirect block mapping table reduction. Inode block mapping reduction increases with more files but is overall negligible for the FileServer workloads. The Impressions workload has more inode block mapping reduction and less indirect block reduction as compared to the FileServer workloads, indicating that it has smaller file sizes. Finally, there is a small part of mappings that FSDV does not remove; most of these mappings are for global file system metadata such as block group description blocks, data and inode bitmaps, and for directory blocks. Overall, we find that indirection mappings can be largely removed.

We also measure the run time of FSDV for these workloads; one of our goals is to have short FSDV run time so that it has less impact on foreground I/Os (*e.g.,* FSDV can be scheduled during device idle time). Figure 6.4 shows the time taken to run FSDV with the FileServer workloads and the Impressions file system image. Overall, we find that the run time of FSDV is small (from 2 to 5 seconds). We

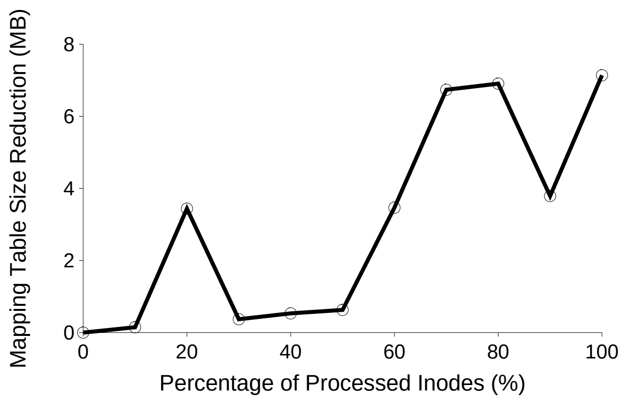


Figure 6.5: **Mapping Table Space Reduction over Different Amount of Inodes.** *This figure plots the amount of mapping table space reduced by the offline FSDV with different amount of inodes for the Impressions workload (I-3.6G).*

further break down the run time into the time spent on processing mappings of data blocks, indirect blocks, and inode blocks, mount and unmount time, and the rest of the time (*e.g.*, time spent on reading block group description blocks). We find that most of the FSDV time is spent on processing data and indirect blocks and such time increases with larger file size and larger file system size.

6.3.2 Impact of Increasing Amount of Processed Inodes

One of the challenges we met in designing FSDV is the way of handling address mappings added by the device. Currently, we handle them by changing the write interface to include the inode number, so that FSDV can process only these changed files and not the whole file system. An alternative to this problem is to let FSDV scan the whole file system image. Thus, it is important to study the effect of reducing the amount of processed files (*e.g.*, by passing the inode number) on mapping table space reduction and FSDV performance.

To study this effect and the cost of whole file system scanning, we change the number of inodes (from 0 to 100% of the total number of inodes) that we process and evaluate the reduced mapping table size and FSDV run time with the file system image generated by Impressions (I1). For each percentage value, we randomly select a set of inodes to process and invoke the offline FSDV after the whole file system image has been written.

Figure 6.5 plots the amount of reduced mapping table space against the number

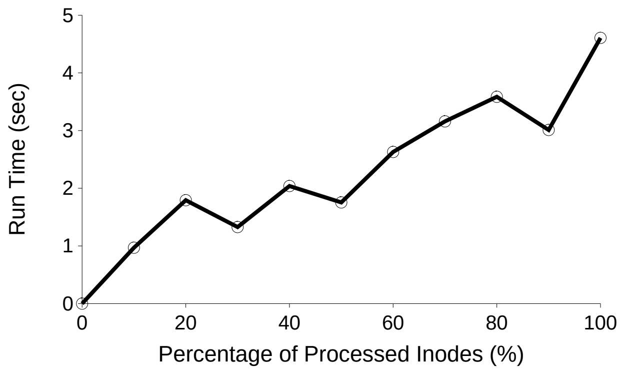


Figure 6.6: **Run Time over Different Amount of Inodes.** *This figure plots the offline FSDV run time with different amount of inodes for the Impressions workload (I-3.6G).*

of processed inodes. Overall, we find that for most of the times, with more inodes processed, more mappings are removed. However, such relationship is not linear. For example, there is a sudden increase from 40% to 60% of the inodes. There is also a sudden increase and drop in the amount of reduced mapping table space at 20% of the inodes. The file system generated by Impressions has a certain file size distribution (which mimics real file systems); certain files can be much bigger than the rest of the files. Because of the randomness in the way we select inodes to process, at 20% FSDV may happen to process one or more of the big files, resulting in a large reduction of mapping table space.

Figure 6.6 plots the time taken to run FSDV with different amount of processed inodes. As the number of inodes increase, the FSDV run time also increases. Different from the mapping table size results, we find that the run time increase is more steady.

Overall, we find that increasing the number of inodes to be processed by FSDV results in more mapping table space reduction but higher FSDV run time. The Impressions file system image that we use only has 3.6 GB data; for a large file system, the time to scan the whole file system can be much higher. Therefore, frequent whole file-system scans by FSDV are not a viable solution; one needs to either reduce the number of inodes to process (our current solution) or increase the frequency of FSDV (our future work).

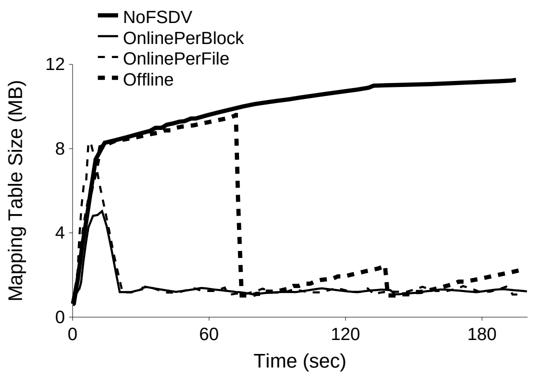


Figure 6.7: **Mapping Table Space Over Time.** *This figure plots the mapping table space change over time running the FileServer F3 workloads with no FSDV, offline FSDV, and per-block and per-file online FSDV.*

6.3.3 Comparison of Different Modes of FSDV

Offline and online are the two options of invoking FSDV. The online FSDV further has two options, per-file and per-block processing. We now present our evaluation results on the difference of these modes.

We first evaluate the mapping table space reduction of different modes of FSDV. In this set of experiments, we repeat the FileServer F3 workload (each running for 60 seconds) and examine the mapping table space change. For the offline FSDV we unmount the file system after each run, invoke FSDV, and then mount the file system. Figure 6.7 plots the mapping table space changes when there is no FSDV running (*i.e.*, normal kernel), when running the offline FSDV, and when running the per-block and per-file online FSDV.

We first find that without FSDV, the mapping table size accumulates as the workloads runs. The initial increase in the mapping table size (time 0 to 5 seconds) is due to the way FileBench runs; it pre-allocates directories and files before running I/Os. With FSDV (both offline and online), the mapping table size decreases and stays low, suggesting that FSDV can dynamically reduce indirection mapping cost.

Comparing the offline and the online modes, we find that the offline FSDV decreases the mapping table size periodically (when it is invoked), while the online FSDV decreases the mapping table size when it first runs and the mapping table size stays low. The online FSDV is triggered by the threshold of mapping table size;

when the mapping table size is above the threshold, the online FSDV is triggered. Therefore, the mapping table size always stays at or below the threshold. Between the per-file and per-block online FSDV, we do not see a significant difference.

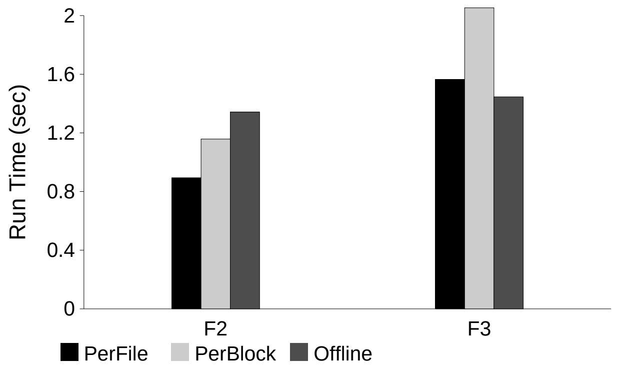


Figure 6.8: **Run Time of Different Modes of FSDV.** *This figure plots the run time of the offline FSDV, per-file online FSDV, and per-block online FSDV with the F2 and F3 workloads. Each run time value is an average of mutiple runs.*

We then evaluate the run time of different modes of FSDV. Here, we use both the FileServer F2 and F3 workloads; F2 has the same number of files as F3 but contains smaller files. Figure 6.8 plots the average run time of the offline, per-file online, and per-block online FSDV with these workloads. We first find that the per-block online FSDV takes longer time to run than the per-file online FSDV; the per-block FSDV exchange information with the device and the file system for each block (*e.g.*, syncing and blocking the block), creating higher overhead than the per-file FSDV, which only does such operations once for each file. Comparing with the offline FSDV, the online modes have longer run time with larger files and file systems (F3), suggesting that the overhead of syncing and blocking data is higher.

For the online mode, one important overhead it causes is the blocking of foreground I/Os; we evaluate such blocked time for both the per-file and per-block online FSDV modes. Figure 6.9 plots the average time that I/Os to a block are blocked because of the per-file or the per-block FSDV. As expected, the per-file FSDV blocks I/Os much longer than the per-block mode, since a whole file is blocked when per-file FSDV is processing a block in it, even though the rest of the file is not being processed. We also find that when file size is larger, the blocked time (with both per-file and per-block modes) is longer.

Overall, we find that the online FSDV allows more dynamism in the mapping

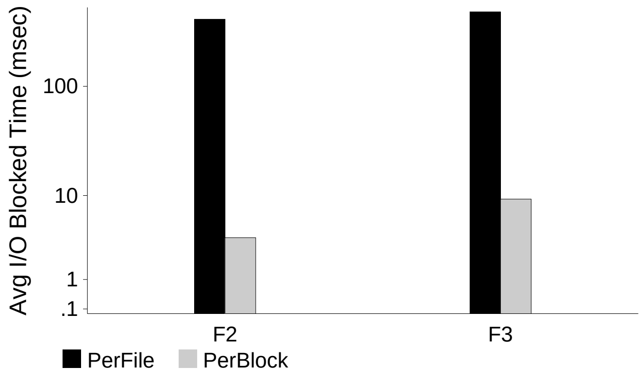


Figure 6.9: **I/O Blocked Time.** *This figure plots the average time a foreground I/O is blocked to a block (in log scale) because of the per-file and per-block FSDV when running the FileServer F2 and F3 workloads.*

table space reduction than the offline mode. The online per-block mode takes longer running time than the per-file mode but requires shorter foreground I/O blocking time.

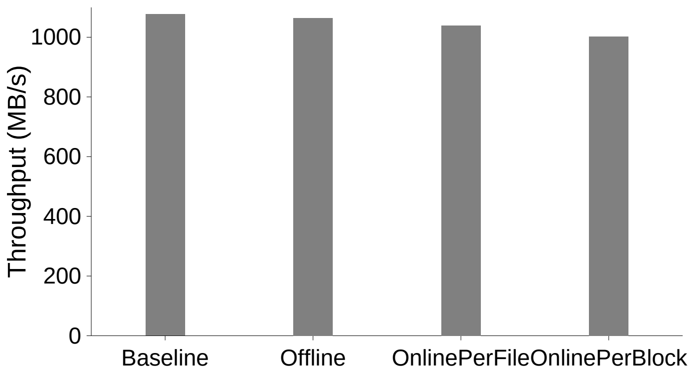


Figure 6.10: **Throughput of Foreground I/Os.** *This figure plots the throughput of foreground I/Os with no FSDV, with offline FSDV, and with per-block and per-file online FSDV when running the FileServer F3 workload.*

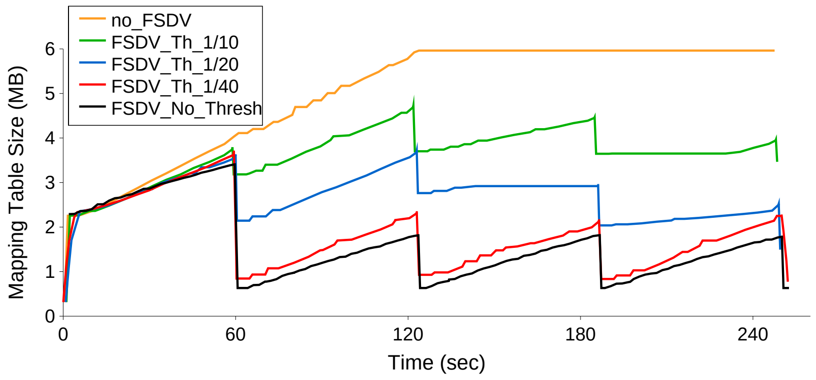


Figure 6.11: **Mapping table size over time**

6.3.4 Overhead on Normal Operations

The impact of FSDV on normal I/Os (*i.e.*, when the FSDV tool is running) is another important metric; one of our goals of FSDV is to have low impact on normal I/Os. We also evaluate the performance overhead of the offline and online (per file and per block) FSDV on normal I/Os with the FileBench macro-benchmark.

Figure 6.10 presents the throughput of normal I/Os when FSDV is not running, using the unmodified Linux kernel, the OS ported to the offline FSDV and the OS ported to the online FSDV (per file and per block). We find that overall, the overhead of FSDV on normal I/Os is low. The overhead in normal I/O operations under the kernel ported to the offline FSDV is mostly due to the FSDV optimization to only process changed inodes; the file system records and sends the updated inodes to the device periodically. In addition to this overhead, the online FSDV also requires file system to sync blocks, causing it to have higher overhead than the offline FSDV.

6.3.5 Optimization Results

Finally, we run one FileServer workload F2 and invoke the FSDV tool periodically (every one minute) to evaluate the effect of FSDV optimization policies. Figure 6.11 presents the mapping table size change over time with basic FSDV, with FSDV using different hot data thresholds, and without FSDV. The hot data threshold is set so that the files that are updated within 1/10, 1/20, and 1/40 of the time window between two FSDV runs are not processed.

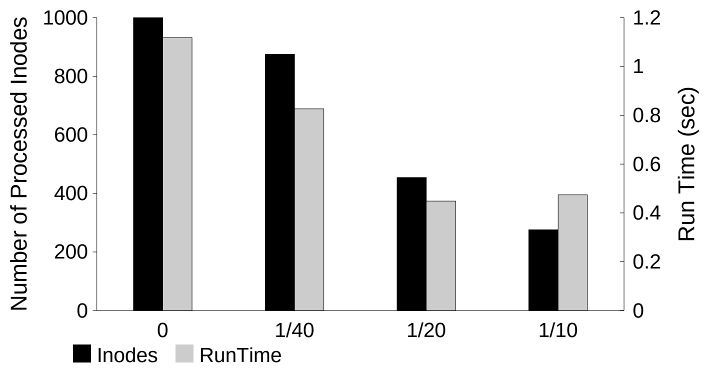


Figure 6.12: **Effect of different FSDV threshold**

We find that the smaller the hot data threshold is, the more mapping table space is reduced; the basic FSDV reduces most amount of mapping table. When hot data threshold is small, more file will be processed. Figure 6.12 shows the run time and number of processed inode with these different hot data thresholds. The run time is the average of all FSDV runs. We find that a lower threshold (fewer inodes ignored) results in more inodes to be processed; thus its run time is also lower. However, the run time of threshold 1/20 and 1/10 is similar because of the fixed run-time cost of FSDV.

6.4 Summary and Discussion

In this chapter, we presented the File System De-Virtualizer, which dynamically reduces the virtualization cost in flash-based SSDs. FSDV removes such cost by changing file system pointers to use device addresses. A major design decision we made is to use separate address spaces, so that blocks can be identified as in different status. We designed several modes of FSDV, offline, online per-file, and online per-block, which have different benefits and costs.

We implemented FSDV as a user-level tool and ported the ext3 file system and the emulated SSD to support FSDV. Our evaluation results demonstrated that FSDV can remove SSD indirection costs significantly in a dynamic way. We also found that there is only a small overhead on foreground I/Os with FSDV.

Comparing with nameless writes, we found that FSDV requires much less change to the OS than nameless writes. The lines of code for nameless writes in the OS is 4370 and is 201 for FSDV. The I/O interface change because of FSDV is also much smaller than nameless writes; only the write command needs to be changed to include the inode number in the path from the OS to the device, while nameless requires fundamental changes to the I/O interface. The change FSDV makes to the write interface can easily be integrated into the ATA interface, since it only changes the forward path from the OS to the device. Finally, FSDV is more dynamic than nameless writes. FSDV can be invoked at any time (*e.g.*, when the memory space pressure in the device is high or when the device is idle), thus causing less overhead to foreground I/Os. Nameless writes are an interface change to all the I/Os, and thus presents an overhead to all foreground I/Os.

Chapter 7

Related Work

This chapter discusses various research efforts and real systems that are related to this dissertation. We first discuss literatures on flash memory and flash-based storage systems. We then discuss other systems that exhibit excess indirection and previous efforts to remove excess indirection. We close this chapter with with other efforts in new storage system interfaces.

7.1 Flash-based Storage

In recent years, flash-based storage have become prevalent in both consumer and enterprise environment, and various new techniques and systems have been proposed for different problems related to flash-memory storage. In this section, we discuss related works on various aspects of flash-based storage.

7.1.1 Flash Memory Management Software

Most flash-based SSDs use a flash translation layer (FTL) to virtualize their internal resources. To reduce the memory size required to store the mapping table for each flash page (usually 2 KB to 8 KB page size), most modern SSD FTLs use a hybrid approach to map most of the data at flash erase block granularity (usually 64 KB to 1 MB) and a small part of page-level mapping for on-going I/Os. A large body of work on flash-based SSD FTLs and file systems that manage them has been proposed in recent years [24, 34, 48, 59, 60].

The poor random write performance of hybrid FTLs has drawn attention from researchers in recent years. The demand-based Flash Translation Layer (DFTL) was proposed to address this problem by maintaining a page-level mapping table and writing data in a log-structured fashion [40]. DFTL stores its page-level mapping table on the device and keeps a small portion of the mapping table in the device cache based on workload temporal locality. However, for workloads that have a bigger working set than the device cache, swapping the cached mapping table with the on-device mapping table structure can be costly. There is also a space overhead to store the entire page-level mapping table on device. The need for a device-level mapping table is obviated with nameless writes and FSDV. Thus, we do not pay the space cost of storing the large page-level mapping table in the device or the performance overhead of swapping mapping table entries.

A different approach to reduce the cost of indirection mapping in SSDs is to move the SSD virtualization layer and the indirection mapping tables from SSDs to a software layer in the host. DFS is one such approach, where a software FTL in the host manages all the address allocations and mappings on top of raw flash memory [46]. With this approach, the cost of virtualization within the device is removed, but such cost (though reduced) still exists in the host. Instead of moving the FTL indirection layer to the host, nameless writes and FSDV remove the excess device indirection and thus do not incur any additional mapping table space cost at the host. Moreover, nameless writes and FSDV both work with flash-based SSDs instead of raw flash memory.

File systems that are designed for flash memory have also been proposed in recent years [41, 92, 93]. Most of these file systems use log-structured allocation and manage garbage collection and flash wears. With such file systems directly managing flash memory, there is no excess indirection or the associated indirection mapping table cost. However, these file systems can only work with raw flash memory. They require knowledge of the flash memory internals such as OOB area size. Operating directly on flash hardware can also be dangerous as we have shown in Chapter 5. Instead, nameless writes and FSDV remove excess indirection by making small changes to existing file systems and flash-based SSDs and thus provide a more generalized solution.

7.1.2 Hardware Prototypes

Research platforms for characterizing flash performance and reliability have been developed in the past [14, 17, 26, 57, 58]. In addition, there have been efforts on prototyping phase-change memory based prototypes [7, 21]. 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 nameless writes 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 [87, 88] and educational purposes [25]. Furthermore, we investigated changes to the flash-device interface, while past work looks at internal FTL mechanisms.

7.2 Excess Indirection and De-indirection

Excess indirection exists in many systems that are widely used today, as well as in research prototypes. In this section, we first discuss a few other systems that exhibit excess indirection besides flash-based SSDs, the focus of this dissertation. We then discuss previous efforts in removing excess indirection.

Excess indirection arises in memory management of operating systems running atop hypervisors [16]. The OS manages virtual-to-physical mappings for each process that is running; the hypervisor, in turn, manages physical-to-machine mappings for each OS. In this manner, the hypervisor has full control over the memory of the system, whereas the OS above remains unchanged, blissfully unaware that it is not managing a real physical memory. Excess indirection leads to both space and time overheads in virtualized systems. The space overhead comes from maintaining OS physical addresses to machine addresses mapping for each page and from possible additional space overhead [2]. Time overheads exist as well in cases like the MIPS TLB-miss lookup in Disco [16].

Excess indirection can also exist in modern disks. For example, modern disks maintain a small amount of extra indirection that maps bad sectors to nearby locations, in order to improve reliability in the face of write failures. Other examples include ideas for “smart” disks that remap writes in order to improve performance (for example, by writing to the nearest free location), which have been explored in previous research such as Loge [28] and “intelligent” disks [89]. These smart disks require large indirection tables inside the drive to map the logical address of the write to its current physical location. This requirement introduces new reliability challenges, including how to keep the indirection table persistent. Finally, fragmentation of randomly-updated files is also an issue.

File systems running atop modern RAID storage arrays provide another excellent example of excess indirection. Modern RAIDs often require indirection tables for fully-flexible control over the on-disk locations of blocks. In AutoRAID, a level of indirection allows the system to keep active blocks in mirrored storage for performance reasons, and move inactive blocks to RAID to increase effective capacity [91] and overcome the RAID small-update problem [75]. When a file system runs atop a RAID, excess indirection exists because the file system maps logical offsets to logical block addresses. The RAID, in turn, maps logical block addresses to physical (disk, offset) pairs. Such systems add memory space overhead to maintain these tables and meet the challenges of persisting the tables across power loss.

Because of the costs of excess indirection, system designers have long sought methods and techniques to reduce the costs of excess indirection in various systems.

The Turtles project [12] is an example of de-indirection in virtualized environment. In a recursively-virtualized environment (with hypervisors running on hypervisors), the Turtles system installs what the authors refer to as *multi-dimensional page tables*. Their approach essentially collapses multiple page tables into a single extra level of indirection, and thus reduces space and time overheads, making the costs of recursive virtualization more palatable.

Finally, we want to point out another type of redundancy and the removal of it: the redundancy (duplication) in data and de-duplication [98, 68, 55]. Different from de-indirection whose purpose is to reduce the space and memory cost of excess indirection, the main purpose of de-duplication is to remove redundant copies of data to save storage space. The basic technique of de-duplication is simple, only one copy of redundant data is stored and multiple pointers to this copy represent the (redundant) copies of the data. Maintaining and accessing such structures can cause overhead, even though the cost has been reduced largely over the past years [98]. In contrast, the technique of de-indirection removes the redundant indirection directly without adding additional metadata (*e.g.*, pointers), and thus does not have the same overhead as de-duplication. The major cost of de-indirection, however, lies in the need to change storage system and interface design.

7.3 New Storage Interface

In this section, we discuss several new types of storage interfaces that are related to this dissertation.

Range writes [8] use an approach similar to nameless writes. Range writes were proposed to improve hard disk performance by letting the file system specify a range of addresses and letting the device pick the final physical address of a write. Instead of a range of addresses, nameless writes are not specified with any addresses, thus obviating file system allocation and moving allocation responsibility to the device. Problems such as updating metadata after writes in range writes also arise in nameless writes. We propose a segmented address space to lessen the overhead and the complexity of such an update process. Another difference is that nameless writes target devices that need to maintain control of data placement, such as wear leveling in flash-based devices. Range writes target traditional hard disks that do not have such responsibilities. Data placement with flash-based devices is also less restricted than traditional hard disks, since flash-based memory has uniform access latency regardless of its location.

In addition to nameless writes, there have been research and commercial efforts on exposing new flash interfaces for file systems [45], caching [30, 43, 67, 78], key-value stores [32], and object stores [49, 50, 96]. 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 in our hardware prototype and propose solutions to overcome them.

While we re-use the existing SATA protocol to extend the SSD interface in our hardware prototype, 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 [72] attach SSDs to the PCI express bus, which allows a driver to implement the block interface directly if wanted. Similarly, the Marvell DragonFly cache [65] bypasses SATA by using an RPC-like interface directly from a device driver, which simplifies integration and reduces the latency of communication.

Chapter 8

Future Work and Conclusions

The advent of flash-memory technology presents both opportunity and challenges. A major issue of flash-based SSDs is the space and performance cost of its indirection.

In this dissertation, we proposed the technique of de-indirection to remove the SSD-level indirection. We started with presenting our efforts in building an accurate SSD emulator in Chapter 3. The emulator was used in later parts in this dissertation and can be used by general SSD-related research. We then presented a new type of interface, the nameless writes, to remove SSD-level indirection in Chapter 4. Next, we discussed our experience with prototyping nameless writes with real hardware in Chapter 5. Finally, in Chapter 6, we presented another method to perform de-indirection, a file system de-virtualizer, which overcomes the drawbacks of nameless writes. We focus on flash-based SSDs as a major use case but the technique of de-indirection is applicable to other types of virtualized storage devices.

In this chapter, we first summarize our de-indirection techniques and emulation and hardware experience in Section 8.1. We then list a set of lessons we learned in Section 8.2. Finally, we outline future directions where our work can possibly be extended in Section 8.3.

8.1 Summary

In this section, we summarize the contributions of this dissertation. We first review the experience of building SSD emulator and implementing new design on real hardware. We then discuss the two methods of performing de-indirection: nameless writes and FSDV.

8.1.1 Emulation and Hardware Experience

We implemented an SSD emulator, which works as a Linux pseudo block device and supports three types of FTLs, page-level, hybrid, and nameless-writing FTLs. To model common types of SSDs with parallel planes, we leverages several techniques to reduce the computational overhead of the emulator. For example, we separate data storage and SSD modeling into different threads. Our evaluation results show that the emulator can model writes accurately with common types of SSDs.

Beyond our efforts in building an accurate SSD emulator, we also built the new design of nameless writes with real hardware. When building the nameless writes hardware prototype, we met a set of new challenges that we did not foresee with emulation. For example, two major challenges are to integrate data in the I/O return path and to add upcalls from the device to the host OS. We proposed a split-FTL approach, which leaves low-level flash operations in the device and runs the bulk of the FTL in the host OS.

Implementing nameless writes in a hardware prototype was a substantial effort, yet ultimately proved its value by providing a concrete demonstration of the performance benefits of the nameless writes design.

Overall, we found that the effort required to implement nameless writes on hardware is comparable to the effort needed to implement an SSD emulator. While we faced challenges integrating new commands into the operating system and firmware, with the SSD emulator we have also struggled to accurately model realistic hardware and to ensure that 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.

8.1.2 De-indirection with Nameless Writes

Our first method to perform de-indirection is nameless writes, a new write interface built to reduce the inherent costs of indirection. With nameless writes, only data and no name (logical address) is sent by the file system to the device. The device allocates a physical address and returns it to the file system for future reads and overwrites.

We met several challenges in designing and implementing nameless writes. First, there is a high performance cost caused by recursive updates to de-virtualize a block. We solve this problem by introducing the separation of address space into logical and physical ones. Another challenge we met is the need for flash-based SSDs to migrate physical blocks requires the physical addresses to be changed in the file system. We used a new interface to upcall from the device to the file system to inform it about the physical address change.

We ported the Linux ext3 file system to nameless writes and built nameless writes with both our SSD emulator and with real hardware. Our evaluation results with both emulation and real hardware showed that nameless writes greatly reduced space costs and improved random-write performance.

8.1.3 File System De-Virtualizer

Our second method to perform de-indirection is the File System De-Virtualizer, which does not require fundamental changes to the OS and I/O interface (a major drawback of nameless writes). FSDV is a light-weight user-level tool which scans file system pointers and change them to use device physical addresses. FSDV can be invoked periodically, when device mapping table is above a threshold, or when the device is idle. We implemented an offline version of FSDV, which requires the file system to be unmounted and mounted before and after the FSDV run. We also implemented an online version, which does not require such detachment of the file system.

To achieve the goal of dynamic de-virtualization, we proposed a new design to separate different address spaces and block status within a file system. A block can use a logical block address which the device maps to its device address, an old physical address which the device maps to its current device address, or a device address with which no mapping is needed. We change the file system to treat bitmap as a tracking of block status in the logical address space and to use a free block counter for allocation.

Our evaluation results of FSDV show that it can remove device indirection cost in a dynamic way with little overhead to foreground I/Os. We also found that FSDV requires much less change to the OS is more dynamic than nameless writes.

8.2 Lessons Learned

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

Excess indirection can be removed

Excess indirection exists in flash-based SSDs. From our experience of nameless writes and FSDV, we find that such excess indirection can be removed. Nameless writes remove the SSD indirection by changing the I/O interface.

FSDV removes the SSD indirection by dynamically reading and changing file system pointers. As a result, both the space and performance overhead of indirection is largely reduced. We believe that such de-indirection techniques can be generalized into other systems that exhibit excess indirection.

Accurate emulation of fast devices that have internal parallelism is difficult

From our SSD emulation experience, we find that implementing SSD models (*i.e.*, different FTLs) is relatively straightforward, while making the emulator work accurately with real systems requires careful thinking and much more efforts.

A major challenge we find in implementing the SSD emulator is to support SSD internal parallelism. To emulate the parallel processing of multiple I/O requests using a single thread is difficult. Currently, we use two threads to separately perform data storage and SSD modeling. Our emulator is accurate enough for the purpose of this dissertation (we focus on writes). However, to emulate faster operations accurately with more parallelism (*e.g.*, the faster read operations with 20 parallel planes), our SSD emulator is not accurate enough; increasing the number of cores used by the emulator can be one solution.

Hardware is different from simulation and emulation

From our hardware experience with building nameless writes on the OpenSSD board, we learned a set of lessons and found that real systems with real hardware is much different from simulation and emulation.

First, we found that the OS stack to a real device (SATA-connected) is more complex than to an emulator. One needs to be careful when integrating new commands in this stack (and different schedulers in the stack).

A bigger problem we met when we implemented nameless writes on real hardware is the difficulty in sending data from the device to the OS both as a return of a file system write and as a upcall initiated by the device.

These and other problems that we met when building the hardware nameless writes prototype were not foreseen when we built nameless writes with the SSD emulator. The lesson we learned in this experience is that one should always have existing real systems in mind when designing new systems.

Interface change is hard

Our initial thought when designing de-indirection methods for flash-based SSDs is that through a simple interface change like nameless writes, de-indirection can be removed easily. However, when we started to build nameless writes with real systems and real hardware, we found that interface change is actually very difficult.

Two major difficulties we met with nameless writes are adding data to the return path of normal writes and augmenting the control path with device upcalls. These operations require significant changes to the ATA protocol and many OS layers, and turned out to be extremely difficult to implement.

Because of these difficulties, we started to think about new solutions for de-indirection and designed the file system de-virtualizer. As compared to nameless writes, FSDV requires only small changes to the OS stack and the I/O interface, all of which can be implemented with real hardware systems. Another advantage of FSDV is that it can dynamically remove the cost of indirection. For example, it can be scheduled at device idle time. Nameless writes, on the other hand, add an overhead to each write.

Our efforts to build new interface with existing systems demonstrated 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 is possible by tunneling them through native commands.

8.3 Future Work

De-indirection is a general technique to remove excess indirection; we believe it can be used in systems other than flash-based SSDs as well. In this section, we outline various types of future work of de-indirection.

8.3.1 De-indirection with Other File Systems

Porting other types of file systems to use nameless writes and FSDV would be an interesting future direction. Here, we give a brief discussion about these file systems and the challenges we foresee in changing them to use nameless writes and FSDV.

Linux ext2: The Linux ext2 file system is similar to the ext3 file system except that it has no journaling. While we rely on the ordered journal mode to provide a natural ordering for the metadata update process of nameless writes in ext3, we need to introduce an ordering on the ext2 file system. Porting ext2 to FSDV on the other hand is straightforward, since FSDV does not require any ordering and does not change the journaling part of ext3.

Copy-On-Write File Systems and Snapshots: As an alternative to journaling, *copy-on-write* (COW) file systems always write out updates to new free space; when all of those updates have reached the disk, a root structure is updated to point at the new structures, and thus include them in the state of the file system. COW file systems thus map naturally to nameless writes. All writes to free space are mapped into the physical segment and issued namelessly; the root structure is mapped into the virtual segment. The write ordering is not affected, as COW file systems all must wait for the COW writes to complete before issuing a write to the root structure anyway.

A major challenge to perform de-indirection with COW file systems or other file systems that support snapshots or versions is that multiple metadata structures can point to the same data block. For both nameless writes and FSDV, multiple metadata blocks need to be updated to de-virtualize a block. One possible way to control the number of metadata updates is to add a small amount of indirection for data blocks that are pointed to by many metadata structures. An additional problem for nameless writes is the large amount of associated metadata because of multiple pointers. We can use file system intrinsic back references, such as those in btrfs, or structures like *Backlog* [63] to represent associated metadata.

Extent-Based File Systems: One final type of file systems worth considering are *extent-based* file systems, such as Linux btrfs and ext4, where contiguous regions of a file are pointed to via (pointer, length) pairs instead of a single pointer per fixed-sized block.

Modifying an extent-based file system to use nameless writes would require a bit of work; as nameless writes of data are issued, the file system would not (yet) know if the data blocks will form one extent or many. Thus, only when the writes complete will the file system be able to determine the outcome. Later writes would not likely be located nearby, and thus to minimize the number of extents, updates should be issued at a single time.

For FSDV, instead of the file tree that uses indirect blocks, the extent tree needs to be processed to use physical addresses, which may break the continuity of the original logical addresses. Therefore, another mechanism is needed that de-virtualizes extent trees.

Extents also hint at the possibility of a new interface for de-indirection. Specifically, it might be useful to provide an interface to *reserve* a larger contiguous region on the device; doing so would enable the file system to ensure that a large file is placed contiguously in physical space, and thus affords a highly compact extent-based representation.

8.3.2 De-indirection of Redundant Arrays

As flash-based storage is gaining popularity in enterprise settings, a major problem to be solve is the reliability of such storage. Redundancy solutions such as RAID [75] can be used to to provide reliability. One way to build reliable and high-performance storage layer is to use arrays of flash-based SSDs [64, 94].

The major problem with de-virtualizing redundant arrays of SSDs is how underlying physical addresses of the devices and their migration (due to garbage collection and wear-leveling) can be associated with file system structure and RAID construction.

Another important issue of de-virtualizing RAID is to maintain the array formation, such as mirroring and striping. Since we do not maintain address mapping after de-indirection, it is difficult to maintain such array formation. For example, a nameless write can be allocated to two different physical addresses on a mirrored pair. Since the file system stores only one physical address of the data, we cannot locate the data on the other mirrored pair. Even if we allocate the same physical address on both pair, one of them can be migrated to a different physical address because of garbage collection or wear-leveling. Similar problems happen with striping and parity, too.

8.3.3 De-indirection in Virtualized Environment

Another interesting environment which can use de-indirection of storage devices is the virtualized environment (*e.g.*, when a guest OS uses a flash-based SSD as its storage device). The virtualized environment provides both opportunities and challenges to perform de-indirection.

With the virtualized environment, the hypervisor has better control and more freedom in its access to various guest states. For example, currently FSDV requires the file system to send the inode number with each block write, while the hypervisor can acquire such information (the inode number) by peaking into the guest memory. With this and other similar techniques performed by the hypervisor, we believe that the guest file system and its device interface will require no or only small changes.

Another situation in the virtualized environment is the use of device indirection layer in software [46]. The major challenge and difference in this situation is to provide dynamic de-indirection; the indirection table space for a guest can be dynamically allocated and changed over time. The hypervisor thus can dynamically remove a certain amount of indirection from one guest. FSDV is an initial effort to provide such dynamism; we believe that better solutions exist to make use of hypervisor and the virtualized environment.

8.4 Closing Words

As software and hardware are getting more complex and are likely to remain so in the future, redundant levels of indirection can exist in a single system for different reasons. Such excess indirection results in both memory space and performance overhead.

We believe that after carefully examining the cause of excess indirection, we can remove the redundant indirection without changing the basic layered structure of an existing system. We hope that this dissertation can help researchers and system builders by demonstrating how redundant indirection can be removed. We also hope that this dissertation serves as a hint for future system designer to be cautious about adding another level of indirection.

Bibliography

[1] A. Modelli and A. Visconti and R. Bez. Advanced flash memory reliability. In *Proceedings of the IEEE International Conference on Integrated Circuit Design and Technology (ICICDT '04)*, Austin, Texas, May 2004.

[2] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[3] N. Agarwal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[4] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, February 2009.

[5] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

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

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

[8] A. Anand, S. Sen, A. Krioukov, F. Popovici, A. Akella, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Banerjee. Avoiding File System Micromanagement with Range Writes. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[9] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and V. Prabhakaran. Removing the costs of indirection in flash-based ssds with nameless writes. In *HotStorage*, 2010.

[10] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.5 edition, 2012.

[11] B. Tauras, Y. Kim, and A. Gupta. PSU Objected-Oriented Flash based SSD simulator. http://csl.cse.psu.edu/?q=node/321.

[12] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. HarEl, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested

Virtualization. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[13] S. Boboila and P. Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

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

[15] J. S. Bucy and G. R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, January 2003.

[16] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

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

[18] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. W. Storer. Mercury: Host-side Flash Caching for the Data Center. In *Proceedings of the 2012 IEEE Symposium on Mass Storage Systems and Technologies (MSST 2012)*, April 2012.

[19] Cade Metz. Flash Drives Replace Disks at Amazon, Facebook, Dropbox. htpp://www.wired.com/wiredenterprise/2012/06/flash-data-centers/all/, 2012.

[20] P. Cappelletti, C. Golla, and E. Zanoni. *Flash Memories*. Kluwer, 1999.

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

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

[23] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[24] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System Software for Flash Memory: A Survey. In *Proceedings of thei 5th International Conference on Embedded and Ubiquitous Computing (EUC '06)*, pages 394–404, August 2006.

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

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

[27] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[28] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 237–252, San Francisco, California, January 1992.

[29] Facebook. Facebook FlashCache. `http://www.github.com/facebook/flashcache`.

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

[31] Fusion-io Inc. ioDrive2. `http://www.fusionio.com/products/iodrive2`.

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

[33] Fusion-io Inc. ioXtreme PCI-e SSD Datasheet. `http://www.fusionio.com/ioxtreme/PDFs/ioXtremeDS_v.9.pdf`.

[34] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37:138–163, June 2005.

[35] E. Gal and S. Toledo. Algorithms and data structures for flash memories. In *ACM Computing Surveys*, 2005.

[36] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, and G. R. Ganger. Timing-accurate Storage Emulation. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[37] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of MICRO-42*, New York, New York, December 2009.

[38] L. M. Grupp, J. D. Davis, and S. Swanson. The Bleak Future of NAND Flash Memory. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

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

[40] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 229–240, Washington, DC, March 2009.

[41] J.-Y. Hwang. F2FS: Flash-friendly file system, 2013. Presented at the Embedded Linux Conference.

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

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

[44] Intel Corporation. Intel X25-M Mainstream SATA Solid-State Drives. `ftp://download.intel.com/design/flash/NAND/mainstream/mainstream-sata-ssd-datasheet.pdf`.

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

[46] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[47] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee. A Group-based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '07)*, October 2007.

[48] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-Based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT '08)*, Seoul, Korea, August 2006.

[49] Y. Kang, J. Yang, and E. L. Miller. Efficient storage management for object-based flash memory. In *Proceedings of the 18th Annual Meeting of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2010)*, August 2010.

[50] Y. Kang, J. Yang, and E. L. Miller. Object-based scm: An efficient interface for storage class memories. In *Proceedings of the 27th IEEE Conference on Mass Storage Systems and Technologies (MSST 2011)*, May 2011.

[51] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory Based File System. In *Proceedings of the USENIX 1995 Winter Technical Conference*, New Orleans, Louisiana, January 1995.

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

[53] Y. Kim, B. Tauras, A. Gupta, D. M. Nistor, and B. Urgaonkar. FlashSim: A Simulator for NAND Flash-based Solid-State Drives. In *Proceedings of the 1st International Conference on Advances in System Simulation (SIMUL '09)*, Porto, Portugal, September 2009.

[54] S. Kleiman. Flash on compute servers, netapp inc. In *HPTS*, 2009.

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

[56] J. Lee, E. Byun, H. Park, J. Choi, D. Lee, and S. H. Noh. CPS-SIM: Configurable and Accurate Clock Precision Solid State Drive Simulator. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC '09)*, Honolulu, Hawaii, March 2009.

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

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

[59] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *In Proceedings of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008)*, February 2008.

[60] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. *IEEE Transactions on Embedded Computing Systems*, 6, 2007.

[61] A. Leventhal. Flash Storage Today. *ACM Queue*, 6(4), July 2008.

[62] S.-P. Lim, S.-W. Lee, and B. Moon. FASTer FTL for Enterprise-Class Flash Memory SSDs. May 2010.

[63] P. Macko, M. Seltzer, and K. A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[64] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential RAID: Rethinking RAID for SSD Reliability. In *Proceedings of the EuroSys Conference (EuroSys '10)*, Paris, France, April 2010.

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

[66] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 238–251, Saint-Malo, France, October 1997.

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

[68] D. T. Meyer and W. J. Bolosky. A Study of Practical Deduplication. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST '11)*, San Jose, California, February 2011.

[69] MjM Data Recovery Ltd. Bad Sector Remapping. `http://www.ukdatarecovery.com/articles/bad-sector-remapping.html`.

[70] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, and F. Trivedi. Bit error rate in nand flash memories. In *Proceedings of the 46th IEEE International Reliability Physics Symposium (IRPS '08)*, Phoenix, Arizona, April 2008.

[71] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the EuroSys Conference (EuroSys '09)*, Nuremburg, Germany, April 2009.

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

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

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

[75] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

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

[77] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

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

[79] M. Saxena, Y. Zhang, M. M. Swift, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[80] M. She. *Semiconductor Flash Memory Scaling*. PhD thesis, University of California, Berkeley, 2003.

[81] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-Based Write Caches. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[82] D. Spinellis. Another Level of Indirection. In A. Oram and G. Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 17, pages 279–291. O'Reilly and Associates, 2007.

[83] Sun Microsystems. Solaris Internals: FileBench. `http://www.solarisinternals.com/wiki/index.php/FileBench`.

[84] Sun-Online. Sun Storage F5100 Flash Array. `http://www.sun.com/F5100`.

[85] S. S. Technology. *Donovan Anderson*. Mindshare Press, 2007.

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

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

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

[89] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[90] Western Digital. NAND Evolution and its Effects on Solid State Drive (SSD) Useable Life. http://www.wdc.com/WDProducts/SSD/whitepapers/ en/NAND_Evolution_0812.pdf, 2009.

[91] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[92] D. Woodhouse. JFFS2: The Journalling Flash File System, Version 2, 2003. `http://sources.redhat.com/jffs2/jffs2`.

[93] YAFFS. YAFFS: A flash file system for embedded use, 2006. `http://www.yaffs.net/`.

[94] Yiying Zhang and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Warped mirrors for flash. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST '13)*, Long Beach, California, May 2013.

[95] Yiying Zhang and Leo Prasath Arulraj and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[96] Youyou Lu and Jiwu Shu and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2013.

[97] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[98] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.