BEYOND THE BLOCK-BASED INTERFACE FOR FLASH-BASED STORAGE

by

Sriram Subramanian

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

Date of final oral examination: 04/26/2012

The dissertation is approved by the following members of the Final Oral Committee:
Andrea Arpaci-Dusseau, Professor, Computer Science
Remzi Arpaci-Dusseau, Professor, Computer Science
Shan Lu, Assistant Professor, Computer Science
Mike Swift, Assistant Professor, Computer Science
Jon Eckhardt, Associate Professor, School of Business

ACKNOWLEDGMENTS

I would like to first thank my advisors, Prof. Andrea Arpaci-Dusseau and Prof. Remzi Arpaci-Dusseau, who have been instrumental in making this journey possible. I was fortunate to take the operating systems class with Prof. Remzi upon getting to graduate school and it eventually led to an RA-ship with the AD group. I initially saw the RA-ship as a resume-builder, but by the end of the spring 2009, I realized I was ready for the long haul. The best part of my initial journey was realizing that I loved working on operating systems and more importantly, the manner in which Prof. Remzi guided me through my indecisiveness (to do a phd or not) led me to believe I could not hope for better advisors. My belief was strengthened by the manner in which they supported my decision to continue my dissertation at Fusion-IO.

Not only have Prof. Andrea and Prof. Remzi helped shape my thoughts, they have also taught me the importance of presenting my ideas clearly and concisely. Though still a work in progress, I have understood the significance of good writing and for that I have to thank Prof. Andrea for her immense patience and clarity. I am grateful to have advisors who shield you from the financial aspects of doing research in the modern world going through one of the worst recessions. Research was meant to be fun and Prof. Andrea and Prof. Remzi helped ensure that consistently. Thinking back, I consider myself extremely lucky: to have been at the right place at the right time.

I would also like to thank Nisha Talagala and Fusion-IO for making my research possible. It was a highly unorthodox setup: allowing a grad-student to work on his thesis while doing something that was somewhat related to what a company was interested. Without Nisha's support, most of the work I did would not have been possible. I am lucky to have received the help of a third advisor, Nisha, who is probably one of the sharpest person I have met. Working at Fusion-io feels like being in grad-school, but with better pay!

As much as people mentioned above have been critical, I would also like to thank Swaminathan Sundararaman. He was not only my roommate for the better part of 5 years, he was also a colleague, a mentor and most importantly, a friend. Throughout the grad-school, I was always able to get reliable and timely advice from Swami. It was useful to have someone who had made the decision to transition from a masters to a phd give you his point of view and this helped me make my mind. He was also instrumental in making the Fusion-io internship possible; without his advocacy, my plans for a doing my dissertation at Fusion would have been impossible. Finally, having a trustworthy friend in a foreign land is extremely important and I was fortunate to have Swami.

Lots of other people have made the last 6 years very eventful and fun. Karthik, who is almost like a brother to me, is one of the most important friends I have. He has been by my side all along and I am very lucky to have someone whom I can trust implicitly. Kaushik is another person who made my life in Madison fun. It was nice to have someone with whom you can have just about any conversation, any time without having to worry about anything.

Before wrapping this section (which I believe is too long, yet grossly incomplete), I would like to thank my sister Shobha and my parents. They were instrumental in letting me pursue my dreams despite the financial hardships they faced. They gave me a good education, a good moral foundation and a chance to succeed.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

BEYOND THE BLOCK-BASED INTERFACE FOR FLASH-BASED STORAGE

Sriram Subramanian

Under the supervision of Professors Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau
At the University of Wisconsin-Madison

The block device interface has shaped the storage industry for almost three decades starting from the tape drives, hard-disks to cdroms. The block interface provides minimalist functionality of reading and writing and this simplicity and broad applicability of the block interface made it a standard. Unsurprisingly, the block interface has not fundamentally changed and the reasons for this goes back to the wide-spread adoption of the standard: the economic incentives not to change the interface outweighs the potential benefits.

The approach of keeping the hardware interface minimal worked well for hard-disks: hard-disks are slow and implementing features in software was efficient. But, the arrival of newer media poses a challenge to this age-old wisdom [19]. Flash can easily outperform traditional spinning disks, but until recently its exorbitant price made it uneconomical. With falling prices, triggered by the massive demand in consumer electronics, flash has suddenly become a very realistic replacement for hard-disks in the form of SSDs. Unfortunately, hiding the raw potential of flash behind a naive block interface is counter productive in more ways than one.

Applications designed in a flash-oblivious manner perform tasks that adversely affect performance and device lifetime. Thus, the top heavy, software-only approach cannot be sustained. In this thesis, we investigate approaches to refactor the software.

In the first part of the thesis, we explore transitioning a standard software feature, namely backups or snapshots, to the flash device. We design and implement ioSnap, which leverages flash to provide fast, file system independent and light-weight snapshots. Through ioSnap, we attempt to understand the tradeoffs of such a transition and the type of support necessary to make such a system feasible.

In the second part of the thesis, we explore new interfaces to the flash device. New interfaces are necessary since we need a generic and systematic approach to expose data management abilities of flash. The new interfaces allow virtualization of address space and time by leveraging the characteristics of the underlying media. We demonstrate the usefulness of the new interfaces by making minimal changes to existing file-systems and applications to add features or simplify existing ones.

Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau

ABSTRACT

The block device interface has shaped the storage industry for almost three decades starting from the tape drives, hard-disks to cdroms. The block interface provides minimalist functionality of reading and writing and this simplicity and broad applicability of the block interface made it a standard. Unsurprisingly, the block interface has not fundamentally changed and the reasons for this goes back to the wide-spread adoption of the standard: the economic incentives not to change the interface outweighs the potential benefits.

The approach of keeping the hardware interface minimal worked well for hard-disks: hard-disks are slow and implementing features in software was efficient. But, the arrival of newer media poses a challenge to this age-old wisdom [19]. Flash can easily outperform traditional spinning disks, but until recently its exorbitant price made it uneconomical. With falling prices, triggered by the massive demand in consumer electronics, flash has suddenly become a very realistic replacement for hard-disks in the form of SSDs. Unfortunately, hiding the raw potential of flash behind a naive block interface is counter productive in more ways than one.

Applications designed in a flash-oblivious manner perform tasks that adversely affect performance and device lifetime. Thus, the top heavy, software-only approach cannot be sustained. In this thesis, we investigate approaches to refactor the software.

In the first part of the thesis, we explore transitioning a standard software feature, namely backups or snapshots, to the flash device. We design and implement ioSnap, which leverages flash to provide fast, file system independent and light-weight snapshots. Through ioSnap, we attempt to understand the tradeoffs of such a transition and the type of support necessary to make such a system feasible.

In the second part of the thesis, we explore new interfaces to the flash device. New interfaces are necessary since we need a generic and systematic approach to expose data management abilities of flash. The new interfaces allow virtualization of address space and time by leveraging the characteristics of the underlying media. We demonstrate the usefulness of the new interfaces by making minimal changes to existing file-systems and applications to add features or simplify existing ones.

1 INTRODUCTION

Storage systems, as we know it, have been designed around the block interface through which the data residing on the media is accessed. The block interface provided minimalist functionality of reading and writing, at the granularity of a hardware-defined block size. The simplicity of the block interface made it popular among mass storage media that supported write-once or write-in-place access (including magnetic tape, hard-disks and optical disks), eventually leading to the standardization of the interface along with the transport protocols for these interfaces [54, 68].

With the overwhelming adoption of the block interface, the software that handled reading and writing blocks became progressively more complex. Users required newer features to help solve their problems. In an attempt to stay hardware oblivious and avoid disturbing a standard that was widely accepted (and in turn also avoid the economic repercussions of said change), new features ended up being pushed to higher-level software.

A large variety of features have been implemented in software to support application demands. A bevy of reliability machinery, such as checksums [14, 122], redundancy [20, 23, 32, 100], and crash consistency techniques [48, 74, 107], have been exclusively implemented in software. Other software features like deduplication [143], versioning [6, 51, 99, 109, 113] and archival [103] have also been implemented in software.

The approach of keeping the hardware interface simple and minimal served the hard-disk era incredibly well. The disks were dumb and slow. Thus, implementing features in software was almost always faster than the drive latency. In other words, the performance bottleneck was the disk and not the application. But, the arrival of newer media poses a challenge to this age-old wisdom [19].

Newer media like flash [108, 115, 126] and persistent memory [30] have been around for a while, but high prices kept them out of mass-adoption. But with the falling prices ($\approx$ \$10 per GB in 1998 [115] to $\approx$ \$1 per GB in 2012 [46]) and growing capacity (from few 100 GBs to TBs), NAND flash is a medium that cannot be ignored anymore.

NAND flash was also packaged behind a block interface in the form of SSDs [56] for rapid adoption. Software designed for hard-disks can natively run on SSDs without any modification and applications can leverage the performance edge delivered by the SSDs. In addition to performance, the power savings enabled by flash-based storage devices makes it an ideal fit for data centers (where total cost of operation is more important than just hardware costs [58]). Thus, the block interface helped in the initial adoption of flash, both in the enterprises and the personal computing markets. Unfortunately, hiding the raw potential of flash behind a naive block interface cannot go on forever.

NAND flash is fundamentally different from hard-disks, both with regards to the technology behind the medium and its performance characteristics. The physics behind NAND flash makes is impractical to overwrite existing content [9]. Thus, NAND flash SSDs adopt log structuring [107] and logs have very different performance characteristics compared to write-anywhere mediums.

The raw performance that is available on NAND flash cannot be fully harnessed by the slow SATA and SCSI busses. SCSI and SATA protocols were designed with the hard-disk in mind (delivering around 600 MBps) and are not suited for flash (requiring in excess of few GBps) [88]. Thus, the first step for high performance, enterprise-grade flash devices was to move to the PCIe bus [40, 55, 82, 130]. By shifting the bottleneck from the hardware to the software, the real problem is exposed: the feature-heavy software.

Applications designed for hard-disks perform tasks that are detrimental to performance as well as device health. For instance, file system block allocation was designed to improve write throughput on hard-disks, but are unnecessary on a log-structured SSD. Application protocols (like data journaling [128], write-ahead logging [84], MySQL double writes [37]), adopted to ensure consistency, are no longer necessary, since the medium is itself a log. Finally, performing more writes than necessary (for example, data journaling) reduces device lifetime: NAND flash devices are designed for a limited volume of writes [12].

One way to alleviate the application bottleneck is to rethink (and hence, refactor) services expected from a flash device. In this thesis, we investigate approaches to deliver well known services in a flash-aware manner. We explore two aspects of this problem: delivering traditional storage features on flash and initiating a rethink of the interface exposed by flash to help write better applications.

In the first part of the thesis, we explore offloading a standard software feature to the flash layer. When it comes to traditional storage features, researchers have explored file systems designed for flash(JFFS [138], NILFS [67], DirectFS [38]), database engines [33, 70] and other applications [43, 45]. Thus, we focus on a relatively unexplored feature in flash, namely backup.

We attempt to address the following question: can snapshots be added to native flash in a high-performance manner. We explore the ability of flash to natively support device-level snapshots. We design and implement ioSnap, which leverages flash to provide fast and lightweight snapshots.

In the second part of the thesis, we adopt a clean-slate approach to address the following question: how can the interfaces exposed by flash be modified to enable applications (old and new) to interact and work on top of flash in a meaningful manner? We introduce the clones system that exposes the ability to virtualize space and time on flash.

Virtualizing space allows interacting directly with the flash translation layer. Traditional storage devices present a static array of blocks with no notion of how data evolved over time. Though the notion of time can be introduced into the storage system (e.g. snapshots), doing so on top of a write-in-place media requires complex copy-on-write software to handle operations correctly. Fortunately, flash is not write-in-place and it is already log structured. More over, the log ordering of data implicitly represents the chronological order. Thus, interfaces to virtualize time helps applications express interest in certain parts of the address space over time.

The clones system provides a set of new APIs that allows applications to interact directly with flash. The following sections elaborate on each of these contributions.

1.1 SNAPSHOTS ON FLASH

Storage system reliability has been a hot topic of research in recent past [14, 48, 74, 94, 100, 107, 122]. Despite these advancements, enterprises continue to backup their data, the reason being both legal [133, 134] and logistical. Enterprises are legally required to keep a copy of their data for a specified period of time and backups help achieve this goal. Moreover, backups help get a system back to a running state after accidental loss of data [15, 124]. Thus, backups are essential in any enterprise.

An important feature common in disk-based file systems and block stores is the ability to create and access *snapshots* [51, 87, 136]. A snapshot is a point-in-time representation of the state of a storage system, and is primarily used in enabling efficient and reliable backup. For this reason, many modern disk-based systems offer snapshots as an important and useful feature [51].

Traditionally, the ability to snapshot relies on copy-on-write (COW) [6, 51, 67, 96] since snapshotted blocks have to be preserved and not overwritten. Moreover, it is a well known fact that SSDs also rely on

COW (through log structuring) to deliver high throughput and manage device wear [9]. Thus, it seems natural for snapshots to seamlessly work with SSD's translation layer (at least in theory, since the log does not overwrite blocks in-place and implicitly provides time-ordering).

In the first part of the thesis, we design and implement ioSnap, which explores the tradeoffs involved in adding the ability to snapshot to a production FTL. We describe the integration of ioSnap into the existing host-based Fusion-io driver and changes to data structures, I/O paths, and the background space scavenger. We then present a careful analysis of ioSnap performance and space overheads. Through experimentation, we show that ioSnap delivers excellent common case read and write performance, largely indistinguishable from the standard Fusion-io VSL. We also measure the costs of creating, deleting, and accessing snapshots, and show that they are reasonable.

1.2 modern interfaces to flash

Standard block devices have been providing the read/write interface for a very long time. Thus, any new device, irrespective of the medium, attempts to first deliver a block interface to encourage adoption. Flash is no exception to this norm having broken into the market first as a solid state device (SSD). SSDs primarily behaves as a block device, with a similar form factor and requiring a standard SCSI or SATA port to connect. More recently, flash vendors have switched to the PCIe bus to maximize the throughput the cards can deliver [40, 82, 130].

Though the block device approach was the right approach to ease the transition into the realm of non-volatile memory, it might not necessarily be the appropriate in the future. Flash devices are fundamentally different from a hard-disk. Besides the fact that the medium on which the devices based are different, there are other more impactful differences. Data on flash is log structured. NAND flash is designed for high throughput and in-place writing could prove detrimental to performance. The second thing to consider is the fact that there exists an indirection layer to hide the log [47, 118]. Finally, flash devices employ garbage collection: a log requires cleanup of old, overwritten data [26, 52, 138].

Despite differences from hard-disks, applications can still work on top of flash and stay oblivious of the fact. But applications designed for hard-disks may eventually prove detrimental to performance and device lifetime. For example, a smart block allocation algorithm in a file system is an overkill on top of a log structured device. Allocating blocks at the file system layer is nullified by the log, which only appends data.

Being a log, flash has the potential to provide simple (yet powerful) transactional interfaces, which applications requiring consistency guarantees can benefit from (for example, file systems and databases can do away their journal or log [64, 84, 127] and move to the atomic write interface [92]). Finally, applications that run on the simple block interfaces may have a negative effect on device lifetime. By doubling writes (to the journal and in-place when in data journaling mode), file systems may reduce a flash device lifetime by half (device lifetimes are directly related to the volume of writes). Thus, applications can not only perform better with more native interfaces to flash, they also indirectly bring down capital expenses by improving device lifetime [95].

Thus, both the applications and the interfaces they use to interact with flash needs a revamp. In this part of the thesis, we explore the new interfaces that are enabled by the log structured nature of the device and its indirection layer. We expose address space virtualization as a native operation on flash that enables simple copying, moving and merging of data. We take this further and leverage these interfaces to virtualize time. This interface supports versioning of data at any granularity (full volumes to a single sector). We finally explore use cases that demonstrate the usefulness of these interfaces. Through small changes to DirectFS [38] and MySQL [121], we were able to implement features like file snapshots, zero-copy file copy, file-level dedup, and atomic writes in MySQL. We also demonstrate the practicality of these new interfaces with a detailed evaluation of the costs of each operation.

A simple block interface not only limits the capabilities of flash, it also can be detrimental to performance and device life. In order to fully realize the potential of flash devices, we need to explore newer, native interfaces. In this part of the thesis, we take the first step towards a more native interface to flash by exposing the ability to virtualize the address space and demonstrate the power of the new interface through multiple use cases.

1.3 CONTRIBUTIONS

The contributions of this thesis are as follows:

- We outline the elements required for an efficient production flash translation layer (FTL) and adds snapshot support in a low-overhead and non-intrusive manner.
- ioSnap's design delivers predictable, high performance with minimal overheads on regular operation. The demands on a production FTL are stringent. It has to have predictable performance while maintaining endurance, and reliability. Our design achieves high performance by keeping the common case (creation, deletion) extremely fast, while deferring work for others (activation).
- We use snapshot-aware rate-limiting algorithms on background threads that perform deferred work. Rate-limiting helps minimize impact on foreground activity.

- We present a case for newer interfaces for flash. Block interface was designed for hard-disks and are insufficient for flash.
- We describe newer interfaces that help realize the true potential of flash. These newer interfaces allow better communications between application and the storage by expressing the intent better. By virtualizing address space and time, complex operations can be simplified. We demonstrate the usefulness of these interfaces by making some simple changes to DirectFS and MySQL.

1.4 OUTLINE

The rest of this thesis is organized as follows.

- **Background:** Chapter 2 provides a brief overview of flash ranging from a description of the physics behind the medium to the types of flash devices. Next, we present a summary of the Fusion-io device driver. The Fusion-io device driver is an enterprise grade production FTL and it forms the backdrop of the rest of the thesis.
- **Snapshots in Flash:** In Chapter 3, we describe the work involved in adding snapshots capability to the Fusion-io driver. We present the design and implementation of ioSnap, which focusses on providing low overhead, flash-aware snapshots. We discuss the tradeoffs involved and evaluate the performance of ioSnap.
- **Modern Interfaces for Flash:** Chapter 4 describes the new interfaces we propose for flash. Newer interfaces allow realization of the full potential of flash and have the ability to improve device lifetime as well. We demonstrate the usefulness of these interfaces by making simple changes to DirectFS and MySQL to add a multitude of new features.

- **Related Work:** In Chapter 5, we present a summary of the related work, to the best of our abilities, in the realm of snapshots, newer flash interfaces, and research efforts that provide similar features as our use cases demonstrated.

- **Future Work and Conclusions:** We discuss some of the future work we envision that would make our interface more powerful and benefit application performance in Chapter 6. Finally, we summarize the highlights of our work and conclude in Chapter 7.

2 BACKGROUND

In modern computer systems (personal computers, smart-phones and, enterprise systems), flash-based storage has become a very important component to deliver high performance as well as energy and space savings. In this chapter, we discuss some of the fundamentals of flash: the types of flash, their characteristics, operations supported and limitations. In the second part of this chapter, we describe the internal details of the Fusion-io block driver. Among the various form factors available for modern flash, Fusion-io produces PCIe-based, enterprise grade flash cards. We present the various components including the FTL, segment cleaner and the crash recovery mechanism.

2.1 FLASH BACKGROUND

Flash is a type of EEPROM (electrically erasable programmable read-only memory). We provide a brief description of the operations supported on flash and the various types of flash memory cells available in the market.

2.1.1 Operations

Typical flash supports a Read, Program and an Erase operation. Bits (one or more) of information can be stored and retrieved from a Floating Gate Transistor (FGT [77]).

Reading a bit corresponds to identifying the charge present in the cell by applying appropriate gate voltage. The absence of a charge results in a current that can be detected by the controller (logical 1). The presence of a charge results in no current (logical 0).

Programming or writing to a cell also involves applying a gate voltage (much higher than the voltage applied during read) and forcing electrons to flow into the gate (FN tunnelling [71]). Programming a "0" is different from programming a "1". The memory cell is at 1 by default and programming a "0" involves forcing electrons into the floating gate.

The **erase** operations is used to reset cells back to the "1" state (i.e., discharge cells), which is accomplished by applying a significantly higher voltage to drain the charge.

The read and program operations take about 15 usec and 68 usec respectively [83] while erasing a page takes up to 100s of msec. Typical flash controllers amortize the cost of an erase over multiple pages.

2.1.2 Classification

Based on the internal organization and the type of gates used, flash may be classified as:

- SLC, MLC, or TLC: A single memory cell or FGT (floating gate transistor [77]) can be used to represent one or more bits of information. The simplest cell (SLC or single level cell) stores only two states: a logical 0 or 1. Thus, the presence of a charge represents logical 0 and absence represents logical 1. MLC or Multi Level Cell can represent 4 states (00, 01, 10 and 11) depending on the amount of charge stored in the cell. By varying the voltage applied during programming, the charge can be controlled, which directly controls the current flowing through the gate during a read. By sensing the current, the logical state of the cell can be determined. TLC or tri-level cell can similarly store up to to 8 states [83, 108].

- NAND or NOR: The two most popular types of flash are NAND and NOR. Both offer very different design points and are used in very different scenarios. NOR flash offers faster read and random access capabilities and thus is preferred for storing code in mobile devices [126]. Unfortunately, NOR flash is extremely slow to write and erase (up to a second [76]). On the other hand, NAND flash provides relatively faster program and erase operations (few microseconds to milliseconds [40]), but does not support random access. NAND flash accesses are at a page granularity. NAND flash is also considerably denser (in theory up to 2x [3]) , which makes it more suitable for large scale deployment.

2.1.3 NAND flash : Characteristics and Limitations

In the rest of the thesis, we only deal with NAND flash due to its large scale deployment and relevance within the enterprise market. In this section, we explore the various form factors of NAND flash in the market and its quantifiable characteristics. We also discuss some of the limitations of commercial NAND flash and the techniques employed to overcome the same.

2.1.3.1 Form Factor and Characteristics

NAND flash is commercially available in a multitude of form factors all the way from the inexpensive consumer grade usb drives to enterprise grade NAS boxes containing flash drives. We present some of the commonly available enterprise grade NAND flash form factors.

Traditionally, NAND flash has been packaged and sold as Solid State Drives (SSDs). SSDs provide the traditional block device interface over SCSI or SATA and share the same form factor as a magnetic hard disk. Manufacturers of SSDs include Intel [56] and OCZ [89]. All the processing is performed inside the SSD and no host resources are consumed for NAND flash related operations. Unfortunately, since the drives are available over SCSI or SATA interfaces, which are connected to slower system buses on the south bridge, SSDs are not highly performant [90]. The cost of such SSDs are at around $1 per GB [125].

In order to fully exploit the performance of a NAND flash drive, newer drives have moved to the PCIe bus instead. Drives like Fusion-io IoDrive [40] and more recently, Micron [82], Intel [55] and Violin Memory [130] leverage the proximity to the CPU by making themselves available on the PCIe bus. These cards may also use an on-load architecture: portions of the device block driver resides on the host memory (e.g. Fusion-io IODrive). The price per GB of these cards are around $1 to $10 per GB.

Finally, some vendors have also started producing stand-alone network attached devices with its primary storage medium being flash (e.g. Fusion-io ION [41], PureStorage [101]). These NAS devices are similar to the standard NetApp or EMC filers with the data available over NFS or CIFS. The NAS boxes also employ RAID schemes to keep the data secure and may also support transparent snapshots and other advanced features [102]. These devices also cost up to $10 per GB of useful data.

Table 2.1 presents a brief summary of the various form factors and their characteristics.

2.1.3.2 Limitations

While NAND flash can deliver significant performance compared to traditional hard-disks, it does come with its own limitations. These limitations are caused by the nature of the material used and the organization of the cells. Some of the well-known limitations of NAND flash are listed below.

Erase cycles

NAND flash, based on the cells used (SLC, MLC or TLC), can only be erased a limited number of times. SLC NAND is rated for around 100K erase cycles, MLC at 5K-10K and TLC at 1K cycles. The process of charging and discharging the cell takes its toll on the material. As

Form Factor	Type	Capa-city	Write Seq. (MBps)	Write Rand. (IOPS)	Read Seq. (MBps)	Read Rand. (IOPS)	Cost $/GB
Intel SSD520	MLC	480 GB	550	70K	520	40K	≈ 1
OCZ Vertex4 (SATA 3.0)	MLC	512 GB	510	85K	560	90K	
Fusion-io IODrive2	MLC	3 TB	1330	140K	1530	110K	≈ 1
Micron P320h (PCIe 2.0)	SLC	350 GB	1120	145K	1840	415K	≈ 1
Fusion-io ION (NAS)	MLC	20 TB	6100	800K	6100	1000K	≈ 5 to 10

Table 2.1: **NAND Flash Device Characteristics.** *The table above presents the device characteristics of the commonly available form factors. The sequential read and write bandwidth are reported in terms of MBps or GBps, while the random read and write are reported in terms of number of 4K IOPS.*

the cells gets worn out, it takes a higher voltage to charge the cells to the same state, thus requiring the firmware to be re-calibrated to work with the old device. The moment a cell fails during a program operation or a read operations, it has to be retired. At times, after a period of dormancy, the cells may be reusable [86].

Access restrictions

NAND flash, unlike NOR, cannot be accessed at random. NOR flash, like standard DRAM, can be read and written to at random over the memory bus. NAND, on the other hand, sits on the IO bus, requiring a larger granularity read and program operation. NAND flash programming is a high voltage, high energy operation and so it is performed at a block granularity [85]. A block consists of a large number of individual pages (e.g. Micron NAND blocks have 64 pages or 128KB [2]).

Write in-place

Write in-place can be very expensive in NAND flash. Updating a single bit in-place is an energy intensive operation requiring a charge or a discharge of a cell depending on its original state. Due to the high voltage applied on the cell, it may cause disturbance in the nearby cells. Overcoming the disturbance also consumes energy [117]. Thus, updates in-place are prohibited. Writes may translate to a (costly) copy-erase-program cycle or writing to a new location and remapping the data.

2.1.3.3 Overcoming NAND Limitations

Most NAND flash vendors use a couple of techniques to overcome the limitations listed above.

Log-style writing

The write in-place restriction is alleviated through log style writing. Data is never written in-place. Instead, in most flash devices, the physical space is structured as a log and data is appended to the head of the log [107]. An indirection table (or the flash translation layer, FTL) is maintained to help construct a notion of a contiguous address space [47, 118].

Almost all NAND flash manufacturers employ an FTL, in one form or the other. FTLs impose space overheads on the device as the mappings in the FTL also have to be persisted [47]. The size of the FTL only grows with updates to the data on the device (fragmentation of address space). Other approaches have attempted to do away with the use of FTLs by delaying logical address allocation until write-completion [142].

Wear leveling

NAND flash cells have limited erase cycles. When data is appended to a log, the old data (i.e., the physical space that is now invalid) can be reclaimed by erasing the page. Once erased, the pages can be added to the free list and reused. Thus, the more often a page is reused, the more erase cycles it would go through making it important to level the wear experienced by each page [81]. By maintaining a count of the number of times a page has been erased, the drive can balance the wear.

2.1.4 Host-based FTLs

Most SSDs employ hardware-based FTLs, which is suitable for the block interface. The FTLs are present within the device, with on-board memory and storage space reserved.

More recently, host-based designs are becoming popular. Host-based FTLs split the FTL logic into two parts: the host-based software and the firmware on the device. The host-based software stores the mapping layer and may also take care of garbage collection. The firmware built into the device is responsible for persisting the data in an efficient manner.

Host-based design makes experimenting with new features practical. Host-based FTLs can be easily modified and redeployed (since most of the interesting features reside in the software running on the host). But, host-based software raises portability and maintainability concerns and comes with considerable cost.

Fusion-io deploys a host-based FTL. Thus, for the remainder of the thesis, we rely on the Fusion-io FTL to act as the backbone for various features we are implementing.

2.2 FUSION-IO VIRTUAL STORAGE LAYER

Fusion-io flash cards are host-based, which means a large portion of the device functionality resides in a driver running on the host (and not on the card).

In this section, we describe the basic operations of the Fusion-io Virtual Storage Layer (VSL) that provides flash management for Fusion-io ioMemory devices. The host based FTL (the VSL driver) aggregates NAND flash modules, performs flash management and presents a conventional block device interface to the user. This opens up a tremendous opportunity to experiment with new features by modifying the driver and not having to tweak a hardware based FTL.

We provide a high-level overview of the FTL; our discussion addresses a previous-generation (though still high performance) FTL, and thus does not necessarily reflect the details of the current generation of Fusion-io devices or drivers.

2.2.1 Log-structured Device

The VSL is log structured since NAND flash does not support efficient overwrites of physical locations (requiring an expensive read, erase and write cycle [9]). Also, the NAND flash chips can be erased and programmed only a certain number of times before the raw bit error rates become too high [46]. To circumvent this costly process, data is never immediately overwritten, but instead appended to the head of the log. Thus, the address exposed by the block device maps to a new address on the log every time the block is modified.

2.2.2 Basic data structures and operations

The two main data structures that manage the copy-on-write operation are the *FTL* and the *validity bitmap* (Figure 2.1 shows a block diagram of the driver). The FTL uses a variant of a B+tree and runs in host memory and translates logical block addresses (or LBA, exposed to higher layers by the block device) to physical addresses (physical

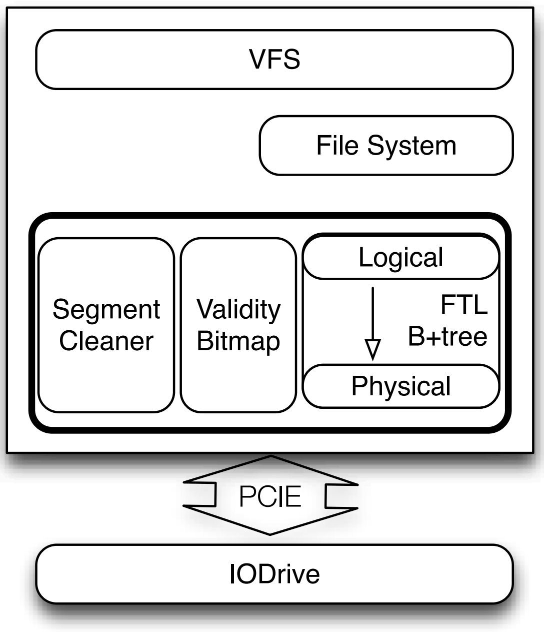

Figure 2.1: **Main Components of the VSL.** *The figure above shows the main components of the Fusion-io VSL device driver. The FTL (also known as the forward map) is a b+tree that translates the logical addresses (exposed to the file system or user) to physical NAND addresses. The validity bitmap is used to track the liveness of physical blocks on the NAND and the segment cleaner is tasked with reclaiming the invalid blocks. As indicated in the figure, the hardware resides over the PCIe bus.*

locations on flash).

The validity bitmap indicates the validity of each physical block on the log. The physical block size is dependent on the format-time block size and this directly impacts the size of the validity bitmap. Block overwrites translate to log appends followed by invalidation of the older blocks. Thus, an LBA overwrite results in the bit corresponding to the old block's physical address being cleared in the validity map and a new bit being set corresponding to the new location as illustrated in Fig. 2.2.

A read operation involves looking up the FTL to translate a range of

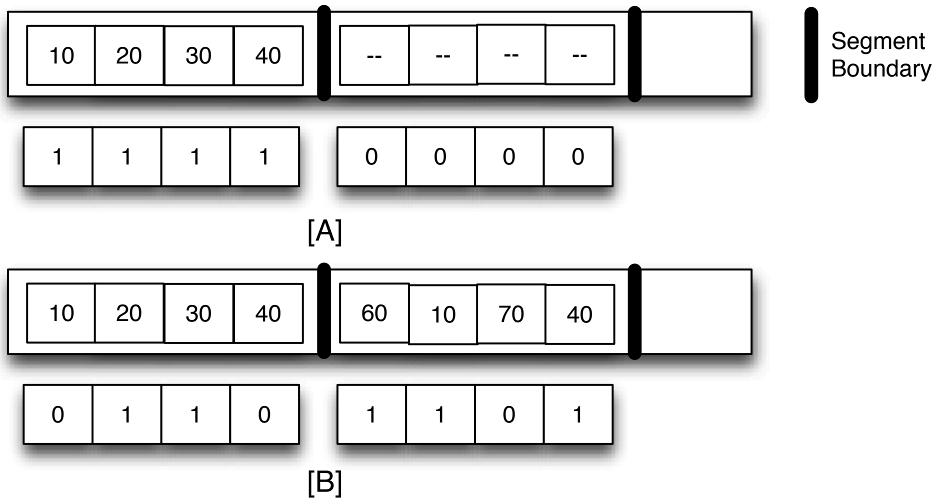

Figure 2.2: **Validity maps.** *The figure above describes the use of validity maps to help the segment cleaner. The example shows a log with 8 blocks spread over two segments. The validity map shown in Fig. A represents 4 blocks 10, 20, 30 and 40. The remaining blocks are unused and the validity bits are cleared. Fig. B represents a point in the future where 4 more blocks have been written to the log namely 60, 10, 70 and 40. As we can observe, blocks 10 and 40 have been overwritten and so the corresponding bits from the validity map are cleared.*

LBAs to one or more ranges of physical addresses. The driver submits read requests for the physical addresses and returns the data back to the requesting file system or user application.

A write request requires a range of LBAs and the data to be written. New blocks are always written to the head of the log. The FTL is used to look up the range of LBAs to figure out if a portion of the write involves an overwrite. If blocks are being overwritten, in addition to modifying the FTL to indicate the new physical addresses, the driver must alter the validity bitmap to invalidate older data and validate the new physical locations.

2.2.3 Segment Cleaning

Segment cleaning is the process of compacting older segments, so as to get rid of all invalid data, thus releasing unused space [107]. Over the life of a device, data on the log is continuously invalidated. Overwrites (and subsequent invalidation) can lead to invalid data being interspersed with valid data. Without the ability to overwrite in place, the driver must erase one or more pages that contain invalid data to regain lost space while copying valid data to ensure correctness. The segment cleaner also modifies the FTL to indicate the new location of the block after moving the data.

The segment cleaner can also have significant performance implications. The erase operation is relatively expensive (erase times are in the order of a few milliseconds for currently available NAND modules), and so must be performed in bulk to amortize latency. The log is divided into equal-sized segments and the segments form the unit of the erase operation. The segment to erase is chosen on the basis of various factors, such as the extent of invalid data contained in the segment and the relative age of the blocks present (degree of hotness or coldness of data). The choice of the segment to clean determines the volume of copy-forward (write-amplification) and how frequently a segment gets erased (wear-leveling), and thus indirectly impacts the overall performance of the system.

2.2.4 Crash Recovery

A clean shutdown of the driver would result in the FTL and validity bitmap being completely de-staged to the log. Upon restart, the metadata can be read from the log and driver state can be restored.

An unclean shutdown would leave the device without some of the metadata (the FTL and the validity bitmaps) needed to run the system, thus requiring reconstruction.

The reconstruction essentially involves a replay of all the writes to the log (in the same order it occurred) and reconstruct the metadata during the replay. But such a replay would require the logical address corresponding to each block of data on the log. Thus, to enable reconstruction in the event of an unclean shutdown, the media stores information about the physical to logical translation for all blocks.

The logical-to-physical mappings are read, processed, and used to create the data structures (FTL and validity bitmap) used for device operation. In an early 2002 version of the driver, the logical to physical translation information is sorted based on logical addresses and log-order. The log-order is necessary to figure out the order of writes to the same logical address: data appearing earlier in the log are overwritten by ones later in the log. Finally, the sorted addresses are used to rebuild the FTL and the validity bitmaps.

3 SNAPSHOTS IN FLASH

An important feature common in disk-based file systems and block stores is the ability to create and access *snapshots* [51, 87, 136]. A snapshot is a point-in-time representation of the state of a storage system, and is primarily used to enable efficient and reliable backup. For this reason, many modern disk-based systems offer snapshots as an important and useful feature [51].

Snapshots are generally implemented at one of two possible layers: the file system layer or the block layer. Block layer snapshots are file system independent and so are easier to deploy and maintain [136]. Unfortunately, they require the help of the file system or application to take consistent snapshots [87]. For example, XFS [123, 139] provides the ability to freeze the file system and allow the volume management system to take a consistent snapshot. Certain file systems like Btrfs [6], ZFS [21, 120] provides the ability to create and access snapshots. Such snapshots are implicitly consistent: the file system can easily track the status of each on-going request and create a snapshot only when it is convinced the state of the file system is consistent.

So the following question arises: when attempting to support snapshots on flash, should we attempt to build a block-layer snapshotting solution or rely on top level file systems? We chose to implement snapshots inside the FTL to leverage the flash-awareness and minimize file system involvement. By residing withing the FTL, we believe, we can provide tighter integration between snapshots and various processes that take place inside the driver. We provide a detailed explanation for this choice in the next section and also address this issue later during evaluation.

In this chapter, we describe the design, implementation, and evaluation of ioSnap,a system that adds flash-optimized snapshots to the

Fusion-io Virtual Storage Layer (VSL). The design and implementation of ioSnap required us to address two significant challenges. Flash is employed by enterprises for its performance. Thus, the first challenge is to ensure performance is not affected in the presence of snapshots. ioSnap is carefully designed to add no overhead to the common I/O paths, thus ensuring that using snapshots does not affect application performance. In addition to this main performance goal, ioSnap also includes novel machinery to rate-limit more expensive background activities (such as snapshot activation), thus ensuring the predictable and high performance customers expect from flash-based products.

The second (and a Fusion-io specific) challenge we face is compatibility. ioSnap is implemented within the Fusion-io device driver. Fusion-io flash cards are host-based, which means a large portion of the device functionality resides in a driver running on the host (and not on the card). Host-based design makes experimenting with new features practical. Significant development effort has gone into achieving VSL's high performance and endurance. Thus, adding new features, however useful, can never come at the expense of user performance, and snapshots are no exception. We thus carefully integrate our snapshot design into the existing VSL I/O paths and data structures, adding little or no overhead and minimizing changes to hardened code.

We describe the ioSnap design and implementation in detail, including its integration into the existing VSL framework and changes to data structures, I/O paths, and the background space scavenger. We then present a careful analysis of ioSnap performance and space overheads. Through experimentation, we show that ioSnap delivers excellent common case read and write performance, largely indistinguishable from the standard Fusion-io VSL. We also measure the costs of creating, deleting, and accessing snapshots, showing that they are reasonable; in addition, we show how our rate-limiting machinery can reduce the impact of ioSnap activity substantially, ensuring little impact on foreground I/O traffic.

The main contributions of ioSnap are as follows:

- Adds snapshot support to a production FTL in a low-overhead and non-intrusive manner.
- Explores a unique design space that focuses on avoiding time and space overheads in common-case operations (data access and snapshot creation, deletion) by sacrificing performance of rarely used operations (snapshot access).
- Novel rate-limiting algorithms to minimize impact of background snapshot work on user activity.
- Careful and thorough analysis depicting the costs and benefits of ioSnap.

The rest of this chapter is structured as follows. First, we provide a detailed description of our motivation behind moving ioSnap into the FTL (Section 3.1). Next, we build on the NAND flash and Fusion-IO device driver background to design and implement ioSnap (Section 3.2). We describe the techniques used to deal with the garbage collector (Section 3.2.6) and crash recovery (Section 3.2.7) among others. Next, we present a detailed evaluation (Section 3.3), followed by a discussion of our results (Section 3.4), and finally conclude (Section 3.5).

3.1 EXTENDED MOTIVATION

The design of ioSnap is based on one major assumption: file systems are not the best place for a flash-aware snapshotting system. In this section, we attempt to provide preliminary evidence to supports the following claim: file systems that are not built ground up for flash perform badly compared to a file system like DirectFS [38] and hence are not suitable for handling more complex functionality like snapshots.

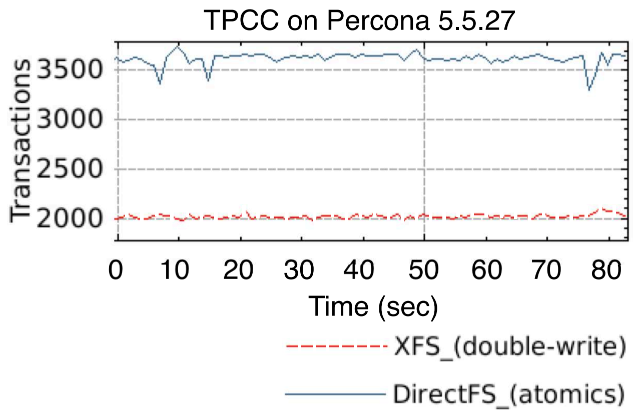

Figure 3.1: **MySQL Performance on XFS and DirectFS.** *The graph above presents a comparison of TPCC performance with MySQL running on XFS and DirectFS.The setup contains two instances of MySQL (Percona 5.5.27) of 700 GB each with 25 GB buffer pool both running on the same physical machine. Both instances run on top of a single 3 TB flash card. The y-axis presents transactions per second with time on the x-axis. As we can observe, DirectFS with atomic writes outperforms XFS by almost 2x. The reason for the massive difference in performance is due to the fact MySQL on XFS needs its logging infrastructure and can safely turn the loggin off with DirectFS.*

3.1.1 Flash Awareness and File Systems

Most file systems are designed for hard disks and unfortunately, this does not necessarily work well on top of flash. In this section, we demonstrate the superiority of a flash aware file system (DirectFS [38]) when compared to spinning media file systems (XFS [123]). DirectFS is a flash aware file system developed at Fusion-io that avoids having a journal and complex block allocation algorithms. DirectFS also supports atomic writes that makes transactional consistency trivial to support. A detailed description of DirectFS appears later in this thesis and it is also available at Josephson et al [62].

We performed a simple test to compare TPCC performance with MySQL on XFS and DirectFS. The setup contains two instances of

MySQL (Percona 5.5.27) of 700 GB each with 25 GB buffer pool both running on the same physical machine. Both instances run on top of a single 3 TB flash drive. Figure 3.1 presents the transactions per second on the y-axis and the time on the x axis. As we can observe, DirectFS with atomic writes outperforms XFS by almost 2x. The reason for the massive difference is due to the fact MySQL on XFS needs its logging infrastructure and can safely turn the logging off with DirectFS. Thus, media awareness is essential to achieving the full potential of flash.

3.1.2 ioSnap: Block or File system level

Most file systems are designed primarily to provide a namespace and operational consistency on top of a simple block interface (of the hard disk). So, file systems continue to treat flash as a block device and perform certain tasks that are not very suitable for flash (like journaling and disk style block allocation). Flash awareness is important as file systems can derive great benefits (performance and complexity) by tailoring their code for flash. Flash has the ability to provide atomic writes [92], making the consistency problem trivial to solve. So having complex machinery like copy-on-write btree [105] or journaling [127] is an overkill for a file system on flash. Thus, flash awareness is important and the lack thereof leads to poor performance as we observed earlier.

We expect the problem will magnify when more functionality (like snapshots) is pushed into the file system. File systems snapshots are unaware of the capabilities and the complexities (segment cleaning, background scrubbing) of the medium underneath and may inadvertently interfere with data management done by the flash driver.

On the other hand, flash, thanks to the nature of the medium, is uniquely capable of delivering snapshots in a manner not possible on disks. Flash devices are log structured and so data is never overwritten. This plays well with snapshots since snapshots are all about retaining relevant older versions of data. The presence of snapshots also impacts the segment cleaner as old snapshotted data cannot be discarded. By implementing snapshots within flash, the segment cleaner, in theory, can help choose the hot and cold segments better: a segment containing snapshotted data is always cold (never modified).

Of course, a block device taking snapshots of user data may result in inconsistent snapshots. A block device would need the file systems assistance in taking consistent snapshots. For example, certain file systems like XFS supports a freeze operation to pause all ongoing reads/writes.

Thus, to keep snapshots flash-aware and generic, we have designed and implemented ioSnap at the block layer, inside the Fusion-io FTL.

3.2 DESIGN AND IMPLEMENTATION

In this section, we describe ioSnap's design and implementation in detail, focusing on the design goals, APIs and individual portions of the VSL that were restructured to work with snapshots.

3.2.1 Design Goals

One of the goals of this thesis was to explore mechanisms to create efficient block-level snapshots in flash. Thus, our primary design goal relates broadly to performance. Users of flash-based storage expect predictable high I/O throughput; thus, any design that sacrifices foreground I/O performance to add snapshot support is not tenable.

A secondary, Fusion-io specific goal deals with integration into the existing VSL. Significant development effort has gone into achieving VSL's high performance and endurance. Thus, it is important that our design integrate effectively within VSL's existing data structures, algorithms and I/O paths.

ioSnap also makes two assumptions about how the snapshots are used. First, it assumes that snapshot creation is common. Modern storage systems are configured to create snapshots every day, hour, and perhaps even every few minutes [29, 66]. We expect this trend to increase with higher throughput devices. Thus snapshot creation is fairly common and must be low overhead.

The second assumption is that snapshot access is (much) less common than creation. Typically, snapshots are activated only when the system or certain files need to be restored from backup. Thus, activations do not match in frequency to creation. Given the expected discrepancy between snapshot creation and usage, it is natural to optimize snapshot creation and defer the bulk of the snapshot work to snapshot activation.

3.2.2 Snapshot API

The basic snapshot operations are minimal: snapshot create, delete, activate and deactivate. Snapshot create and delete are straightforward, allowing the user to persistently create or delete a snapshot (ioSnap implicitly retains all snapshots unless explicitly deleted). Activation is the process of making a snapshot accessible. Not all systems require activations. We make activation a formal step because ioSnap defers snapshot work until activation in many cases. Finally, ioSnap also needs to provide the ability to deactivate a snapshot. The snapshot APIs are meant to provide a set of mechanisms, on top of which workload-specific creation and retention policies may be applied.

Putting it all together, the typical use case may look something like this. The user performs I/O on the device, while periodically creating snapshots, say one every hour. Old snapshots are backed-up and deleted continuously. Occasionally, the user realizes that she needs to restore a few deleted files and has to activate an old snapshot. More-

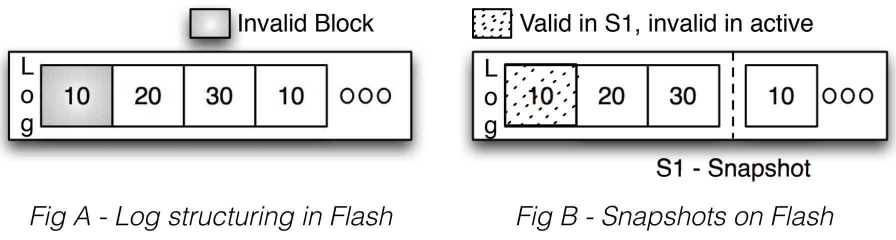
Invalid Block
Valid in S1, invalid in active
Log
10
20
30
10
ooo
S1 - Snapshot
Fig A - Log structuring in Flash
Fig B - Snapshots on Flash

Figure 3.2: **Snapshots on the log.** *This figure illustrates how a log-based device can naturally support snapshots. Each rectangle represents a log, with each square indicating a block, with the logical block address (LBA) mentioned inside. In Fig. A, we show a log with four blocks with LBAs 10, 20, 30 and one block overwritten at address 10. The key aspect is the presence of the invalidated block on the log (until garbage collection). The log in Fig. B, shows how the log can be leveraged to implements snapshots. If we were to write blocks at addresses 10, 20 and 30 and then create a snapshot (call it S1, indicated by the dashed line) and overwrite block 10, then according to snapshot S1, the original block at address 10 is still valid while the active log only see the new block at address 10. Thus by selectively retaining blocks, the log can naturally support snapshots.*

over, the old snapshots have to be moved out of flash to free space and this operation also requires snapshot activation. Following activation, the user may have to mount the file system present in the snapshot to perform any file operations on the device. The user may unmount the file system and deactivate the snapshot later.

3.2.3 Log Structuring

The log-structured nature of flash devices naturally supports snapshots. New blocks written are appended to the head of the log, leaving behind the invalidated data to be reclaimed by the segment cleaner. Thus, supporting snapshots requires selectively retaining older blocks and allowing the segment cleaner to discern snapshot blocks from invalid blocks. Fig. 3.2 shows a detailed example.

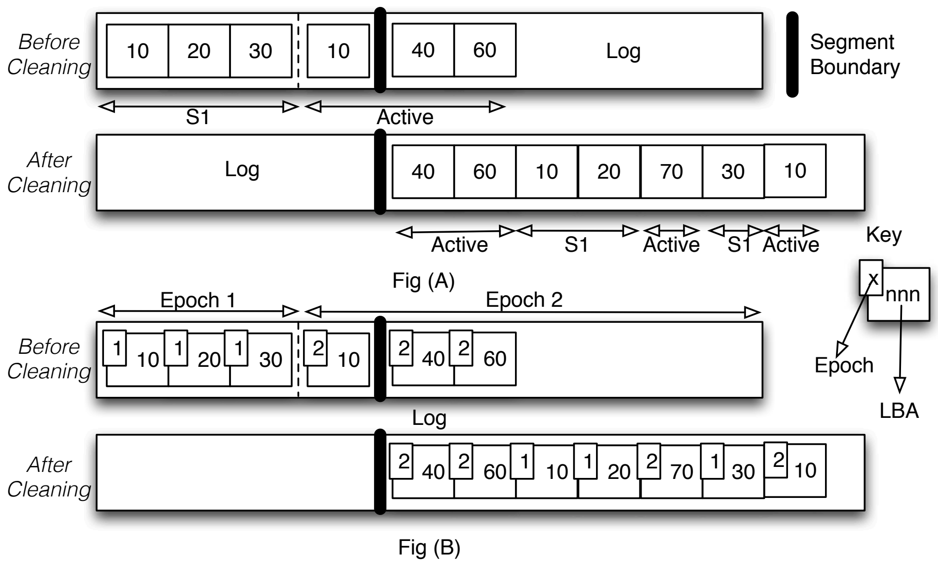

Figure 3.3: **Epochs.** *The figure illustrates the impact of the garbage collector on the time order of the log and the need for Epochs. In Fig. A, we write blocks at LBAs 10, 20, 30, create a snapshot S1 and continue writing blocks to LBAs 10, 40 and 60. The segment boundaries are indicated by the thick black line. While blocks are moved, newer writes may also be processed resulting in active data (LBAs 40,60,10 and 70) getting mixed with blocks from S1 as shown by the log at the bottom of Fig. A and makes distinguishing blocks impossible. Fig. B shows the use of epochs to delineate blocks belonging to various snapshots. Epoch numbers are assigned to all blocks as indicated by the number in the small box on the left top of each block.*

3.2.4 Epochs

Log structuring inherently creates time-ordering within the blocks and the time ordering of blocks can easily allow us to associate a group of blocks to a snapshot. Unfortunately, the segment cleaner moves valid blocks to the head of the log and disrupts the time-ordering. Once blocks from distinct time frames get mixed with active writes, it becomes hard to distinguish blocks that belong to one snapshot from another. Fig. 3.3(A) illustrates the impact of segment cleaning on a sample log. Simply put: when blocks are moved, active data can be mixed with blocks from older snapshots.

In order to implement snapshots in a log-structured medium, ioSnap leverages the notion of *Epochs* [96, 103]. Epochs divide operations (writes) into log-time based sets that segregate operations that took place between subsequent snapshot operations. Every epoch is assigned an epoch number that is a monotonically increasing counter. The log maintains the current epoch number in its block header (i.e., OOB or out of band area) and new blocks written to the log are assigned the current epoch number. By associating every block with an epoch number, ioSnap can (loosely) maintain the notion of log-time despite intermixing induced by the segment cleaner. Fig. 3.3(B) shows how associating epoch numbers to blocks can help manage the block mixing situation. Epochs are incremented when snapshots are created or activated. Every snapshot is associated with an epoch number that indicates the epoch number of blocks that were written after the last known snapshot operation.

3.2.5 Snapshot Tree

A snapshot created at any point in time is related to a select set of snapshots created before. The moment a snapshot is created, every block of data on the drive is (logically) pointed to by two entities: the *active tree* (the device the user issues reads and writes to) and the snapshot that was created. In other words, the active tree implicitly inherits blocks from the snapshot that was created. As subsequent writes are issued, the state of the active tree diverges from the state of the snapshot until the moment another snapshot is created. The newly created snapshot captures the set of all changes that took place between the first snapshot and the present. Finally, as more snapshots are created and activated, ioSnap keeps track of the relationships be-

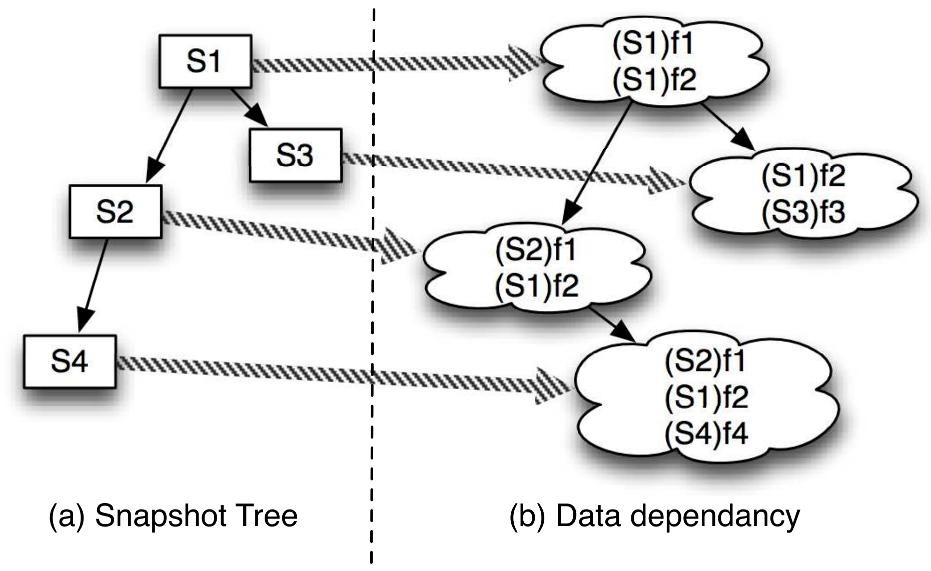

Figure 3.4: **Snapshot tree.** *The figure illustrates the relationship between snapshots based on data they share. The tree to the left shows the relationship between snapshots and the tree to the right shows the data associated with each snapshot (as files for simplicity with the snapshot it originated from in parentheses). Snapshot S1 has two files, f1 and f2. After updating file f1, snapshot S2 is created followed by S4, which adds a file f4. S4's state contains files from S1 and S2. At some point, S1 was activated and file f1 was deleted and a new file f3 was created. Deleting file f2 does not affect the file stored in S1. Activating S1 creates a fork in the tree and after some changes, S3 is created.*

tween the snapshots through a snapshot tree [10] Fig. 3.4 illustrates an example depicting four snapshots and their relationships.

3.2.6 Segment Cleaner

The segment cleaner is responsible for reclaiming space left by invalidated data. Not surprisingly, adding snapshots to the VSL necessitates changes to the cleaner.

3.2.6.1 Implementation Challenges

There are two major issues that the cleaner must address to operate correctly in the presence of snapshots. Originally, as mentioned in Section 2.2.3, the cleaner simply examines the validity bitmap to infer the validity of a block. Unfortunately, when snapshot data is present on the log, a block that is invalid in the current state of the device may still be used in some snapshot. The first challenge for the segment cleaner is thus to figure out if there exists at least one older snapshot that still uses the block, as seen in Fig. 3.3.

Second, in the presence of snapshots, the choice of segment to clean needs to be rethought, the reason being the fact that the segment cleaner has a tendency to mix blocks belonging to various epochs and we would ideally like to keep the degree of intermixing minimal (we do not address the policies required for segment cleaning in the presence of snapshots). By keeping intermixing minimal, snapshot activation can be made more efficient.

3.2.6.2 Copy-on-write validity bitmap

Validity bitmaps are used to indicate the validity of a block with respect to the active log. In the presence of snapshots, a block that is valid with respect to one snapshot may have been overwritten in the active log, thus making maintenance of validity bitmaps tricky.

ioSnap solves the validity bitmap problem by maintaining bitmaps for each epoch (similar to bitmaps in WAFL [51]). During the course of an epoch, the validity bitmap is modified in the same manner as described in Section 2.2.2. When a snapshot is created, the state of the validity bitmap at that point corresponds to the state of the device captured by the snapshot. Having copies of the validity bitmap for each snapshot allows us to know exactly which blocks were valid in a snapshot.

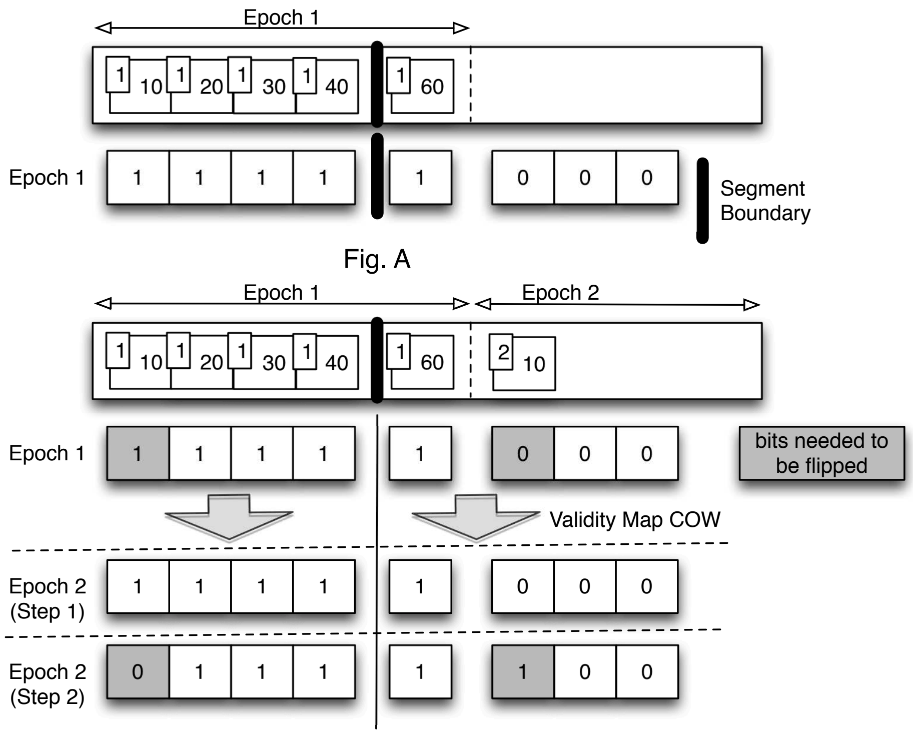

Fig. B

Figure 3.5: **Epochs and Validity maps.** *The figure describes the use of per epoch validity maps to help the segment cleaner. The example shows a simple case with two epochs spread over two segments. The validity map named Epoch 1 corresponds to the validity map of the log at snapshot creation. The validity map in Fig. A shows 5 valid blocks on the log. After snapshot creation, block 10 is overwritten, involving clearing one bit and setting another. The bits that need to be cleared and set are shown in gray in Fig. B and clearly these belong to Epoch 1 and hence read-only. Thus, the first step is to create copies of these validity blocks (shown in step 1). Next, the bits in the newly created epoch 2 are set and cleared to represent the updated state.*

A snapshot's validity bitmap is never modified unless the segment cleaner moves blocks (more details in Section 3.2.6.3). Like epochs, the validity bitmap for a snapshot inherits the validity bitmap of its parent to represent the state of the blocks inherited. The active log can continue to modify the inherited validity bitmap.

The naive design would be to copy the validity bitmap at snapshot creation time, which would guarantee a unique set of bitmaps that accurately represent the state of the blocks belonging to the snapshot. Clearly, such a system would be highly space inefficient since every snapshot would require a large amount of memory to represent its validity bitmaps (for example, for a 2 TB drive, 512 MB per snapshot with 512 byte blocks).

Instead ioSnap takes an approach that relies on copy-on-write for the validity bitmap blocks. It leverages the fact that time ordering of the log typically guarantees spatial collocation of blocks belonging to the same epoch. So, until blocks are moved by the segment cleaner to the head of the log, the validity bitmaps corresponding to an epoch would also be relatively localized. ioSnap uses this observation to employ copy-on-write on validity bitmaps after a snapshot operation where all validity bitmap blocks are marked COW. When an attempt is made to modify a block marked COW, a copy is created and linked to the snapshot (or epoch) whose state it represents. The validity bitmap pages that were copied are read-only (until reclamation) and can be de-staged to the log. Fig. 3.5 illustrates an example where validity bitmaps are inherited and later modified by the active log.

3.2.6.3 Segment Cleaning: Putting It All Together

Segment cleaning in the presence of snapshots is significantly different from the standard block device segment cleaner. The segment cleaner needs to decide if a block is valid or not and the validity of a block cannot be derived just by looking at one validity bitmap. The following steps need to be followed to clean a segment:

Merge validity bitmaps: The validity information for each block is spread across multiple validity bitmaps. To obtain a global view of the device across epochs, ioSnap merges the validity bitmaps (logical OR) and produce a cumulative map for the segment. Epochs corresponding to deleted snapshots need not be merged unless there exists at least one descendant epoch still inheriting the validity bitmap. Fig. 3.6 illustrates this process. Fig. 3.6(A) shows the state of the drive before segment cleaning and Fig. 3.6(B) shows the state of the drive after segment cleaning. Fig. 3.6(C) illustrates an example with a deleted snapshot: Epoch 1 has been deleted and so it will not contribute blocks to the merged map. While handling deleted snapshots, the validity maps corresponding to the epoch may or may not be merged. If there exists at least one child epoch (of the deleted epoch) that still relies on the validity information of the deleted epoch, then we must merge the validity information from a deleted epoch.

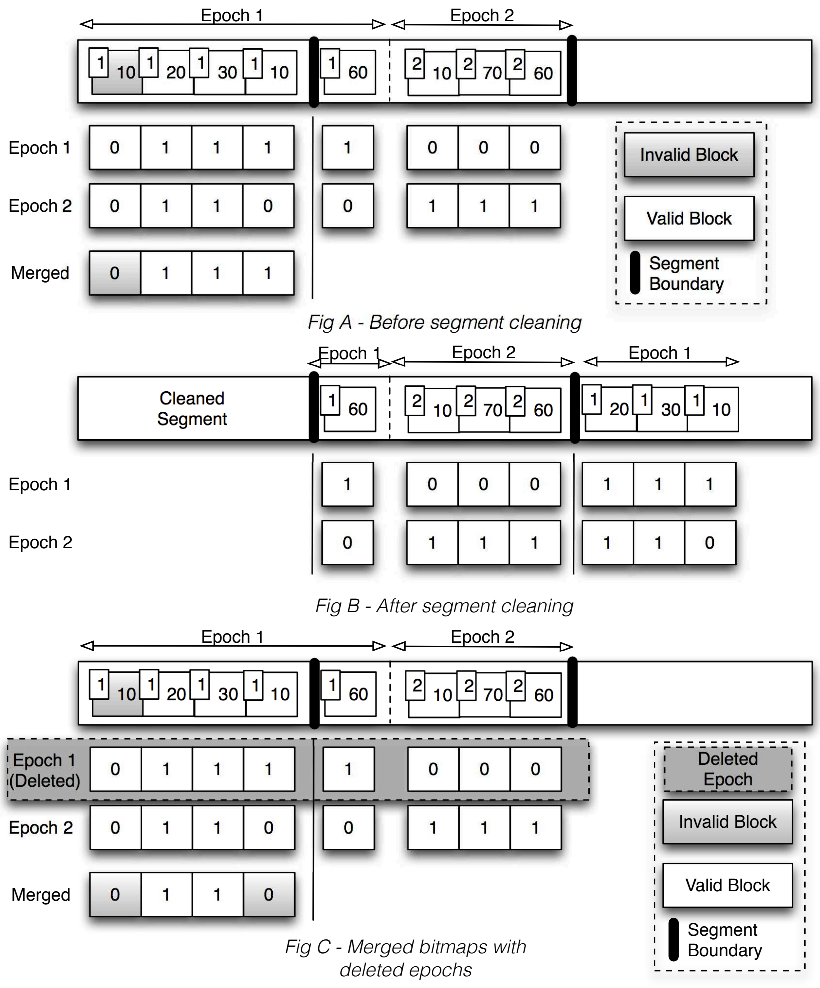

Figure 3.6: **Segment cleaner in operation.** *The figure illustrates the segment cleaner operation in the presence of snapshots. Fig. A and B show the state of the drive before and after segment cleaning. The log in Fig. A uses two segments with 4 blocks each and has one snapshot (indicated by the epochs 1 and 2) with the corresponding validity maps shown below. The segment cleaner merges the validity maps to create the merged map. Based on the validity of blocks in the merged maps, blocks 20,30,10 (from Epoch 1) are copy forwarded to Segment 3 as shows in Fig. B. While moving blocks forward, the segment cleaner has to re-adjust the validity maps to reflect block state in Segment 3. Finally, Fig. C shows an example with a deleted epoch (Epoch 1). The merged bitmap is equivalent to the only valid epoch, Epoch 2. This automatically invalidates blocks from Epoch 1.*

Check for validity of blocks: The merged validity bitmap for a segment represent the globally valid blocks in that segment. The blocks that are invalid are the ones that have been overwritten within the same epoch or the ones that may belong to deleted snapshots.
Move and reset validity bits: For every valid block copy-forwarded, the validity bits need to be adjusted. One or more epochs may refer to the block, which means ioSnap needs to set and clear the validity bitmap at more than one location. In the worst case, every valid epoch may refer to this block, in which case ioSnap may be required to set as many bits as there are epochs.

3.2.7 Crash Recovery

The device state needs to be reconstructed after a crash. In the version of the driver we use, the device state is fully checkpointed only on a clean shutdown. On an unclean shutdown, some elements of in-memory state are reconstructed from metadata stored in the log.

3.2.7.1 FTL Reconstruction

The FTL of the active tree is reconstructed by processing logical to physical block translations present in the log. In our prototype, ioSnap only reconstructs the active trees and does not build trees corresponding to all the snapshots. This goes back to the design choice we made early on to keep operations on the active tree fast and attempting to reconstruct snapshot FTL would impose both space and time overheads.

The reconstruction of the FTL in the presence of snapshots is integrated with the standard crash recovery mechanisms in the driver. The recovery process occurs in two phases and we describe what has been added to each of these phases in order to recreate snapshot state. In the first phase, the snapshot are identified (using snapshot creation notes in the log) and the snapshot tree is constructed. Once the snapshot tree is constructed, ioSnap has the lineage of the active tree whose FTL needs to be constructed. In the second pass, ioSnap selectively processes the translations that are relevant to the active trees. The relevant blocks for the active tree are those that belong to the epoch of the active tree or any of its parent in the snapshot tree. ioSnap also processes the snapshot deletion and activation blocks in the second pass and update the snapshot tree. Once all blocks are processed, ioSnap sorts the entries on their LBA and reconstruct the FTL in a bottom up fashion.

3.2.7.2 Validity-bitmap Reconstruction

Validity bitmap reconstruction happens in multiple phases. In the first phase, ioSnap sorts FTL entries (in block headers) based on the epoch in which the data was created (sorting is required due to the segment cleaner, which disrupts log-time ordering). ioSnap then eliminates duplicate (or older) entries and uniquely identify blocks that are valid and active in the epoch under consideration. Once the valid entries are identified, it constructs the validity bitmap for each epoch starting from the root of the snapshot tree until it reaches the leaf nodes in a breadth first manner. The final validity map for each epoch is reconstructed by merging the epochs that could contribute blocks to this epoch (namely, the parent epochs). While merging, ioSnap eliminates duplicates or invalidated entries. The snapshot tree is traversed in a breadth first manner and validity maps constructed for every epoch in the manner described above. The number of phases needed to reconstruct the validity bitmap depends on the number of snapshots created in the device.

3.2.8 Snapshot Activation

In order to access a snapshot, the user needs to activate it. The primary reason for an activation operation is due to the fact that ioSnap only maintains the mappings of the active tree prior to an activation since keeping multiple FTLs in memory (even under copy-on-write) is prohibitively expensive. Moreover, having multiple maps may require multiple updates to the map when a shared block is moved by the segment cleaner.

Upon activation, ioSnap produces a new writable device that resembles the snapshot (but never overwrites the snapshot). Thus, activation also results in creation of a new epoch to absorb all the blocks written to the new block device. The steps to construct the FTL of an activated snapshot are the same as that of FTL reconstruction of the active tree. The only difference is that the reconstruction starts from the epoch number of the snapshot that needs to be activated instead of the epoch number of the active tree.

3.2.9 Rate Limiting

Snapshot activations and segment cleaning can impact foreground I/O performance. Specifically, activating a snapshot requires reading block headers (i.e., OOB data) from the log and recreating the FTL; similar background traffic is inherent in segment cleaning. This traffic competes with foreground I/Os for device bandwidth and as a result could introduce jitters (or spikes) in foreground workloads, which is unacceptable.

To alleviate the problem, ioSnap rate-limits background I/O traffic during snapshot activation and segment cleaning. Background activation traffic is rate-limited by providing a tunable knob that determines the rate at which snapshots are activated at the expense of activation time. In the case of segment cleaning, ioSnap improves the vanilla rate-limiter by providing a better estimate of the amount of work that needs to be done to clean the segment as default structures do not account for snapshotted data.

3.2.10 Implementing Snapshot Operations

Given our understanding of the log, epochs, and the snapshot tree, we now describe the actions taken during various snapshotting activities. For *snapshot creation*, the following four actions take place. First, the application must quiesce writes before issuing a snapshot create. Second, ioSnap writes a snapshot-create note to the log indicating the epoch that was snapshotted. A note is a special data block that records a specific metadata operation. Third, it increments the epoch counter, and finally, add the snapshot to the snapshot tree.

Snapshot deletions require two steps. First, ioSnap synchronously write a snapshot-delete note to the log. The presence of the note persists the delete operation. Second, it marks the snapshot deleted in the snapshot-tree. This prevents future attempts to access the snapshot. Once a snapshot is marked deleted, the blocks from the epoch corresponding to the snapshot are eventually reclaimed by the segment cleaner in background as described earlier in Fig. 3.6. Thus, deleting a snapshot does not directly impact performance.

The process of *snapshot activation* is more complicated as a result of our design decision to defer work to the rarer case of old-snapshot access, and requires five steps. First, ioSnap validates the existence of the requested snapshot in the snapshot tree. Second, ioSnap synchronously writes a snapshot-activate note to the log. The note ensures accurate reconstruction in the event of a crash by ensuring that the correct tree would be reconstructed to recreate the state of the system. Third, it increments the epoch counter. Activating a snapshot creates

	Vanilla (MB/s)	ioSnap (MB/s)
Sequential Write	1617.34 ± 1.63	1615.47 ± 5.44
Random Write	1375.16 ± 84.6	1380.46 ± 88.9
Sequential Read	1238.28 ± 10.8	1240.51 ± 0.24
Random Read	312.15 ± 1.05	310.23 ± 0.71

Table 3.1: **Regular Operations.** *The table compares the performance of vanilla FTL driver and ioSnap for regular read and write operations. We issued 4K read and writes to the log using two threads. Writes were performed asynchronously and 16 GB of data was read or written to the log in each experiment (repeated 5 times).*

a new epoch that inherits blocks from the aforementioned snapshot. Fourth, ioSnap reconstruct the FTL and validity bitmap as described in Sec 3.2.7 Though our design permits for both readable and writable snapshots, we have only prototyped readable snapshots. *Snapshot deactivation* only requires writing a note on the log recording the action.

3.3 EVALUATION

In this section, we evaluate ioSnap in terms of the impact on user performance during regular read/write operations, snapshot operations, and segment cleaning. We also evaluate the impact on crash-recovery. All experiments were performed on a quad core Intel i7 processor, with a 1.2 TB NAND Flash drive, 12 GB of RAM, running Linux 2.6.35, and a older generation of the VSL driver. The flash device was formatted with a sector size of 4KB (unless otherwise explicitly stated). The overall development effort in implementing ioSnap involved around 4500 lines of code added and 200 existing lines modified, with a large portion of code changes required to handle the copy-on-write of the validity bitmaps during regular operations as well as during segment cleaning.

Operation	Time (usec)	Metadata (KB)
Creation	50	4
Deletion	50	4

Table 3.2: **Snapshot Operations: Create and Delete.** *The table presents the time spent creating and deleting a snapshot. We issued a large number of 4KB writes to the device and after a few seconds, we issued a snapshot creation call. After a while, we deleted the snapshot. In both cases, we measure the time spent during snapshot creation and deletion and found them to be around 50 usec, which is equivalent to the time spent writing a snapshot note on the log.*

3.3.1 Regular Operations

An important goal of ioSnap was to have minimal performance impact on the VSL driver. We employ micro-benchmarks to understand the impact of snapshot support in ioSnap during regular operations. We used sequential and random read/write benchmarks to measure the performance of the vanilla VSL driver and ioSnap. Table 3.1 presents the result of our micro benchmark. From the table, we see that the performance impact of running ioSnap when there are no snapshot-related activity is negligible. This is in tune with our design goal of being close to the raw device performance.

3.3.2 Snapshot Operations

The design of ioSnap introduces multiple overheads that may directly or indirectly impact user performance. Users may observe performance degradation (lower throughput, higher latencies) or increased memory consumption due to snapshot-related activity in the background. We now discuss the implication of snapshot operations on user performance and VSL metadata.

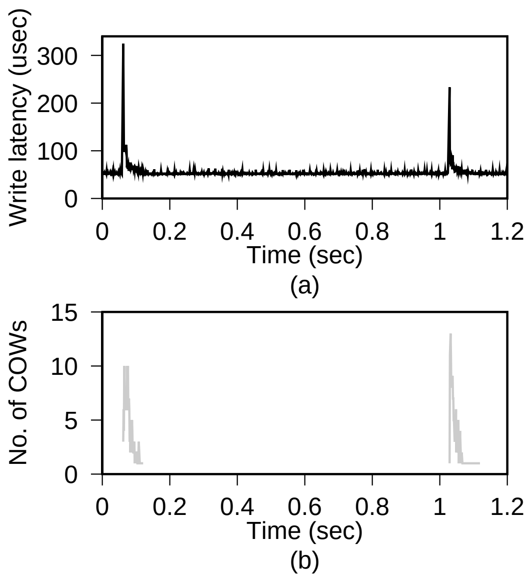

Figure 3.7: **Impact of snapshot creation.** *The figure above illustrates the impact of a snapshot creation operation on the latency observed by the user. In this experiment, we first write 512 byte blocks to arbitrary logical addresses (total data of 3GB and not shown in the figure). Then, at time t=0 sec, we created a snapshot and then continue writing blocks to random locations (of 8 MB). The process of creating a snapshot and writing 8 MB was repeated after about 1 sec into the benchmark. We depict the latency by the solid gray line on the primary Y axis and the number of validity bitmap copy-on-write occurrences with the solid black line on the secondary Y axis.*

3.3.2.1 Creation and Deletion

Snapshot creation and deletion are invoked more frequently compared to activation. These two operations have to be extremely fast to avoid user-visible performance degradation. To measure the performance of snapshot creation and deletion, we ran micro benchmarks described in Table 3.1 and varied the amount of data before the create or delete was issued. In all our experiments, we observed a latency of about 50 μsec and the metadata on log was 4KB (Table 3.2). This is due to the fact that only a snapshot creation (or deletion) note is written to the log and is independent of the amount of data written to the log. The metadata block of 4KB per snapshot is insignificant (on a 1.2 TB drive).

Though snapshot creation is extremely fast, it could impact the performance of subsequent writes. After a snapshot is created, requests that overwrite existing data result in the corresponding validity bitmap to be copied (Section 3.2.6.2). On the other hand, read operations are not impacted after snapshot creation and snapshot deletes do not impact read and write performance.

We wrote a micro benchmark to understand the *worst case* performance impact of COW. In our micro benchmark, we first wrote 3GB of data to the log to populate the validity bitmaps. Then, we created a snapshot (at time t=0 sec) and issued synchronous 512 byte random-writes of 8MB data to overwrite portions of the 3GB data on the log; we repeated the same process again (at time t=1 sec). Also, for this experiment, we formatted the device with 512 byte sectors to highlight the *worst case* performance overheads. Fig. 3.7 illustrates the impact on user observed write latency.

Latency:

From Fig. 3.7a, we can see that the write latencies shoot up (to at most 350 µsec) for a brief period of time ($\approx$ 50 msec) before returning to the pre-snapshot values. The period of perturbation obviously depends on the amount of copy-on-write to be performed; which, in turn, depends on the total amount of data in the previous epochs. We observe similar behavior upon the creation of a second snapshot (at around t=1 sec). Note that this is the *worst case* performance (validity bitmap copy for every write) and the latency spike would be *smaller* for other workloads.

Space Overheads:

Snapshot creation directly adds a metadata block in the log (described earlier) and indirectly causes metadata overheads due to additional validity bitmap pages created as part of COW (bitmaps are kept in-memory). The validity bitmap COW overhead is directly proportional to both data present on the log and data overwritten to distinct logical addresses after a snapshot. Fig. 3.7b displays the count of the validity bitmaps that were copied during the execution of the micro benchmark described above. In the experiment, we observed 196 validity bitmap blocks being copied after the first snapshot incurring an overhead of 784 KB per snapshot (or about 0.024% per snapshot). Note that the validity bitmap COW overheads will decrease with larger block sizes as fewer validity bitmaps would be copied.

In summary, snapshot creation and deletion operations are lightweight (50 usec) and adds a metadata block in the log. But snapshot creation could also impact subsequent synchronous write performance due to COW of validity bitmap pages. The amount of validity bitmaps created is directly proportional to data on the log and the distinct logical addresses that got overwritten.

3.3.2.2 Activation

Snapshot activation requires a scan of the device to identify blocks associated (with the snapshot being activated) and followed by the recreation of the FTL. The log scan and FTL recreation incur both memory and performance overheads. Also, foreground operations could be impacted due to the activation process. We now evaluate the cost of activating a snapshot.

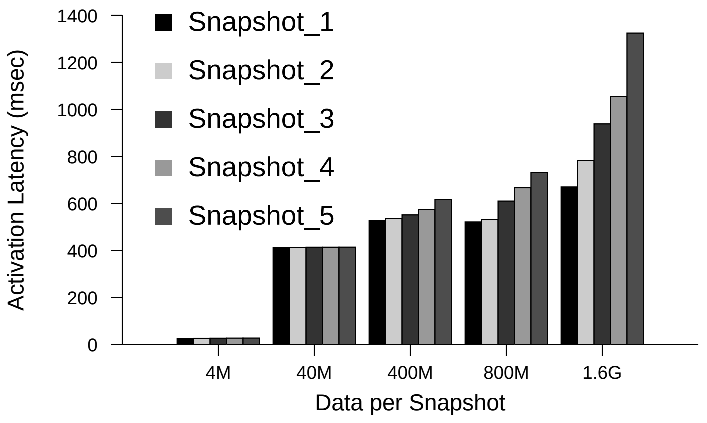

Figure 3.8: **Snapshot activation latency.** *The figure illustrates the time spent in activating snapshots of various sizes. Each cluster represents a fixed volume of data written between snapshots. For e.g., the first cluster labeled 4M represents creation of five snapshots with 4MB of data written between each snapshot create operation. The five columns in each cluster indicates the time taken to activate each of the five snapshots. Within each cluster, every column (i.e, snapshot) requires data from all the columns to its left for its activation.*

Time overheads:

The time taken to activate a snapshot directly depends on the size of the snapshot. To measure snapshot activation time, we first prepare the device by writing a fixed amount of data and then create a snapshot. We then repeat the process for 4 more times (i.e., 5 snapshots are created with equal amount of data). The amount of data written to the device was varied between 4M and 1.6GB. Fig. 3.8 and present the time spent in activating snapshots of various sizes. In this figure, each column within the cluster indicates the time spent in activating a specific snapshot (for example, the first column corresponds to the activation of first snapshot).

From the figure, we make two important observations. First, the more data on the log, the longer it takes to activate a snapshot. Second, more time is required to activate snapshots that are deeper (i.e., they are derived from other snapshots) in the snapshot tree. The increase in activation time is due to the fact that to create the FTL of a snapshot, all of its ancestors in the snapshot tree have to be processed in a top-down manner (Section 3.2.8).

Upon looking at the time spent at various phases of activation (Fig. 3.9), we observed that for fixed log size, irrespective of the number of snapshots present, the time taken to identify blocks associated with a snapshot is constant. The constant time is due to the fact that the segment cleaner could have moved blocks in the log and hence, the entire log needs to be read to ensure all the blocks belonging to the snapshot are identified correctly. For example, to read a log containing 8 GB of data (the column named 1.6 GB), we spend around 600 msec scanning the log followed by the reconstruction phase that may vary from 60 msec (snapshot 1, no dependencies) to 750 msec (snapshot 5, dependent on all 4 snapshots before it).

Space overheads:

Activation also results in memory overheads as we need to create the FTL of the activated snapshot. To measure the in-memory overheads, we first created a log with five snapshots each with 1.6 GB worth of data (using random 4K block writes). We activate each of the 5 snapshots and measure the in-memory FTL size of the activated snapshot. Table 3.3 presents the results.

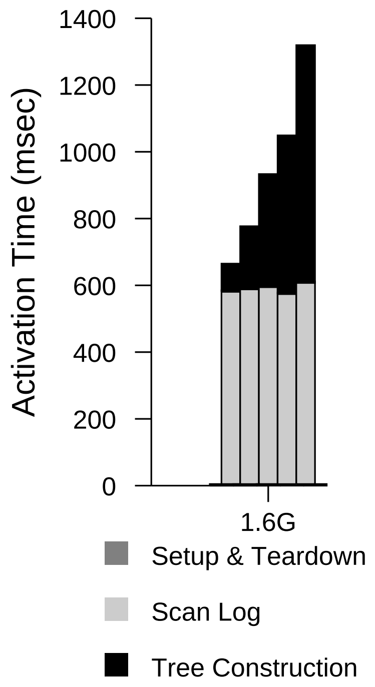

Figure 3.9: **Phases during Snapshot Activation.** *The figure illustrates the time spent during various phases of activating snashots. The workload is identical to the one used in Figure 3.8 and we have shown the breaddown of the various stages for the 1.6 GB cluster. The three main phases are: Setup/Teardown, Log Scan and Tree construction. As we can observer from the figure above, the scan phase is fixed for a given size of the log (8 GB) and the tree construction phase depends (almost linearly) on the depth of the snapshot in the tree.*

From the table, we make two observations: first, with an increase in data present in the snapshot, the memory footprint of the new tree also increases. Second, the tree created by activation tends to be more compact than the active tree with exactly the same state. The reason for better compaction is due to the fact that the original tree is fragmented, while the tree created by the activation is as compact as the tree can be.

Snapshot no. activated	Size of tree at snapshot creation (MB)	Size of tree after snapshot activation (MB)
1	1.38	0.84
2	4.41	3.63
3	7.91	7.09
4	11.20	10.51
5	14.44	13.72

Table 3.3: **Memory overheads of Snapshot activation.** *The table above presents the memory overheads incurred when snapshots are activated. In the experiment we created five snapshots with 1.6 GB of of random 4K block writes being issues between each subsequent snapshot create operations. After each snapshot create operation we measure the size of the tree in terms of the number of tree nodes present. After five snapshots are created, we activate each of those snapshots and measure the in-memory sizes of newly created FTL.*

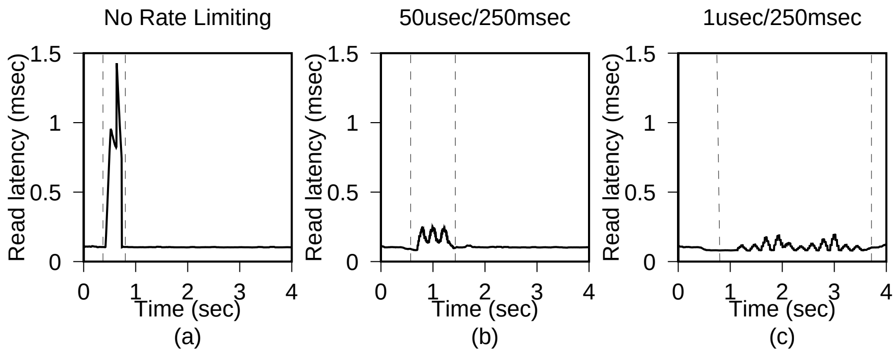

Figure 3.10: **Random read performance during activation.** *The figure above illustrates the performance overhead imposed by activation on random read operations and how rate-limiting activation can help mitigate performance impact. We perform random read operations of 4K sized blocks on the drive with 1 GB of data spread across two snapshots. About 500 msec into the workload we activate the first snapshot that contains 500 MB of data. The random reads average about 100 usec before activation. The naive performance is show in (a) and two rate-limiting schemes are shown in (b), and (c). In each of the rate-limiting scheme, the parameters shown as "x usec/y msec", meaning for every x usec of activation work done, the activation thread has to sleep for y msecs. The dashed lines in each plot indicates the time when activation started and completed.*

Impact on foreground requests:

Activating a snapshot may also interfere with on-going operations. To quantify the interference, we performed a simple experiment where we issued 4K random reads with 1 GB of data spread over two snapshots. Fig. 3.10 shows the results of the experiment. From the figure, we see that the random reads, which average 0.1 msec before activation, shoots up during activation, and finally stabilizes. From Fig. 3.10a, we observe that latency has gone up by almost 10x. Such latency spikes are unacceptable for many performance sensitive enterprise applications.

We implemented a simple rate-limiting scheme to control the amount of work done by for activation. In rate-limiting, we trade-off activation time in favor of user latency (for every **'x'** usec of work done, activation is forced to sleep for **'y'** msecs, expressed as 'x usec / y msecs'). Figs 3.10b (50usec/250msec) and 3.10c (1usec/250msec) show the effect of rate-limiting on activation process. From these figures, we can clearly see that the impact on read performance falls significantly with rate-limiting (worst case read latency decreases from 10x to 2x) but the time to activate increases (from .3 sec to 3.5 sec). The variation in latency could be further reduced by increasing the activation time.

The important thing to note here is the ability of ioSnap to distinguish between foreground and background tasks (namely, regular operations and snapshot activation). If snapshots were implemented on top of flash, say in a file system like btrfs, then activations are fast since btrfs keeps a separate btree for the snapshot. Attempting to use snapshotted data may trigger a large volume of reads that corresponds to the snapshot metadata that was paged out. But the underlying driver has no idea on how to differentiate between regular blocks and snapshot metadata. Thus, designing ioSnap withing the FTL makes sense.

No. of Snapshots	Overall Time (sec)	Validity Merge (msec)
Vanilla (0)	10.42	113.07
0	10.48	127.9
1	10.14	140.65
2	10.8	205.15

Table 3.4: **Overheads of segment cleaning.** *The table presents the overheads incurred while reclaiming a segment. In our experiment, a foreground thread issues 4K random writes filling up multiple segments (around 5 GB of data). Also, a background thread creates snapshots at arbitrary intervals. We force the cleaner to pick up the segment that was just written in order to measure the overheads. We present the overall time spent in cleaning the segment and the time spent in merging validity bitmaps when computing validity of data.*

3.3.3 Segment Cleaning

The segment cleaner copy-forwards valid blocks from a candidate segment. The amount of valid data may increase in the presence of snapshots since blocks invalidated in a snapshot may still be valid in its ancestors. The increase in data movement could impact both segment cleaning time and foreground user requests. To measure the impact of segment cleaning, we use a foreground random write benchmark with 4K block size that writes 5GB of data spreading across multiple segments while a background thread creates two snapshots (still within the first segment). Once the first segment is full, we force the segment cleaner to work on this segment while foreground writes continue to progress.

Table 3.4 show the overheads of snapshot-aware segment cleaning in ioSnap. In the presence of snapshots, the cleaner has to perform additional data movement to capture snapshotted data (568, 628 and 631 MB respectively for vanilla/zero, one and two snapshots respectively). We do not consider the additional data as an overhead. Interestingly, the segment cleaning time neither increases with the number of snapshots nor with the amount of valid data moved. The vanilla VSL driver has rate-limiting built into it where the segment cleaner aggressively performs work when there is more data to move. Upon closer inspection of various stages of the segment cleaner, we observed that with more snapshots present within a segment, the validity bitmap merging operation tends to get more expensive. Even in the presence of zero snapshots, we incur an overhead of 12% and this progressively grows.

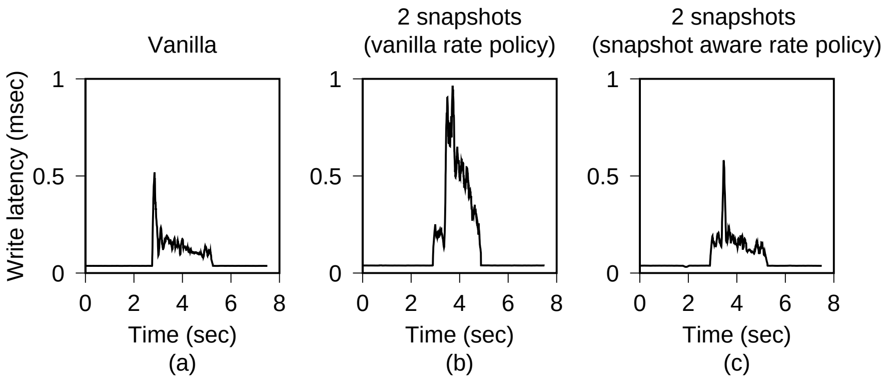

Figure 3.11: **Impact of Segment Cleaner on user performance.** *The figure illustrates the performance overhead imposed by segment cleaning on foreground random writes and how snapshot-aware rate-limiting can help mitigate performance impact. We perform random writes of 4KB blocks on the drive with 5 GB of data spread across two snapshots. The impact on random read performance due to vanilla segment cleaner is show in (a), ioSnap with two snapshots in (b), and ioSnap with snapshot aware rate-limiting in (c).*

Data movement introduced by the segment cleaner imposes overheads on the application visible performance and Fig. 3.11 illustrates this impact on the aforementioned user workload. From the Fig. 3.11a, we can see that vanilla driver does impact the write latency, which is an artifact of the version of the driver we are using. We observed similar behavior in ioSnap with zero snapshots (not shown in the figure).

Fig. 3.11b shows that ioSnap increases write latency by a factor of 2. We mitigate the increase in latency by introducing snapshot awareness to the rate-limiting algorithm in the segment cleaner by providing a better estimate of the number of valid blocks in the segment containing snapshotted data. As we can see in Fig. 3.11c, the overheads are brought back to the original levels shown in Fig. 3.11a.

The ability to rate limit the segment cleaner in a smart manner is one of the biggest gains of having ioSnap withing the FTL. ioSnap can accurately measure the extra data present in snapshots and rate limit the segment cleaner in a more controlled manner. Instead, if snapshots were implemented on top of the FTL, it would automatically become the job of the snapshotting file system to provide explicit information when snapshotted data is not needed anymore. File systems manage the logical addresses: freed logical blocks are reclaimed and reused, but that does not translate to freeing the physical space on the device. Without explicit TRIMs to free physical space, the segment cleaner would have to move more blocks than necessary. Moreover, the file system has no idea about the physical organization of data on flash. This effectively makes it impossible to have snapshot-aware segment cleaning policies. Thus, for these two reasons, putting ioSnap within inside the driver makes more sense.

3.3.4 Crash Recovery

Crash recovery is needed when the flash device is not shutdown cleanly. The driver needs to reconstruct its in-memory state consisting of the FTL and validity maps (Sections 2.2.4 and 3.2.7). We evaluate the cost of recreating the VSL state, specifically measure the overheads of snapshot awareness.

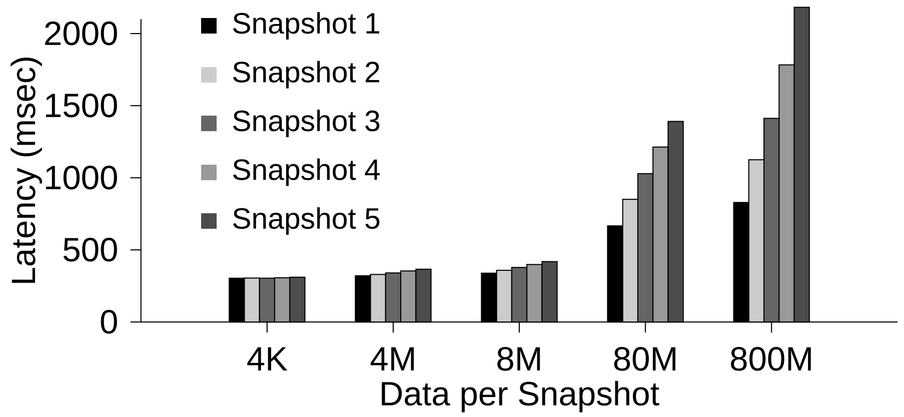

Figure 3.12: **Snapshots and crash recovery.** *This figure shows the time spent recovering from a crash in the presence of snapshots. Each experiment involved writing a fixed amount of data followed by a snapshot. For example, the first cluster involved writing 4KB of data followed by creation of a snapshot and the process is repeated (up to 5 times). The first column within the cluster indicates the time to restart the system with one snapshot and 4K of data and so on.*

Each experiment involved writing a fixed amount of data (4KB to 800MB) followed by a snapshot. For example, the first cluster involved writing 4KB of data followed by creation of a snapshot and the process is repeated (up to 5 times). The first column within the cluster indicates the time to restart the system when a snapshot was created after 4KB was written, the second column represents the time to restart with 8KB (4KB + 4KB bytes) data interspersed by two snapshots and so on. Fig. 3.12 shows the results of our experiment. Clearly, with more data on the log, crash recovery gets longer. A large fraction of the time is spent in scanning the log. The scan time is comparable to the base-driver. Finally, with increasing number of snapshots, the reconstruction phase also grows longer (proportionally) as every snapshot requires its validity map to be constructed. Depending on the depth of the snapshots, the reconstruction time may be almost as long as the scan-time. Thus, this raises the need for more other techniques to alleviate the slow restart times.

3.4 DISCUSSION AND FUTURE WORK

The design choices we made have ensured VSL's performance is not compromised in the presence of snapshots. Snapshot creation is very fast and largely invisible to foreground work. Unfortunately, deferring snapshot operations comes at a price. Clearly, activation is an expensive process in terms of space and time. In the worst case, activations may spend tens of seconds reading the full device and in the process consume memory to accommodate a second FTL tree (that shares no memory with the active tree). Rate-limiting can help control impact on foreground processes (both during activation and segment cleaning) making these background tasks either fast or invisible, but not both.

ioSnap's design choices represent just one of many approaches one may adopt while designing snapshots for flash. Clearly, some issues with our design like the activation and COW overheads need to be addressed. Most systems allow instantaneous activation by always maintaining mappings to snapshots in the active metadata [6, 51, 103, 118], but this approach might not be flash-friendly: any metadata that is not absolutely necessary for the correctness of operation is an overhead and increases cost per GB of flash. Activation overheads may be reduced by precomputing some portions of the FTL and checkpointing on the log. The segment cleaner may also assist in this process by using policies like hot/cold [107] to reduce epoch intermixing, thereby localizing data read during activation. Finally, keeping snapshots on flash for prolonged durations is not necessarily the best use of the SSD. Thus, schemes to destage snapshots to archival disks are required. Checkpointed (precomputed) metadata can hasten this process by allowing the backup manager identify blocks belonging to a snapshot.

3.5 CONCLUSIONS

ioSnap successfully brings together two copy-on-write based techniques, log structuring and snapshotting, in a lightweight and flash-aware manner. The two most important goals of ioSnap were performance and compatibility. ioSnap imposes negligible overhead during regular operations and fits right into the FTL driver. Moreover, any file system running on top of the Fusion-io block device can create very low overhead snapshots.

Given the log structured nature of the FTL, one could easily assume adding snapshots must be trivial, which unfortunately is far from the truth. Some of the lessons we learned in the process of designing and implementing ioSnap are as follows.

- Host based FTLs can impose severe overheads on the system memory. Thus, any approach that keeps this overhead minimal is always better. This was particularly true with respect to snapshots: keeping copy-on-write copies of all snapshot FTLs in memory is not feasible.

- Activations in ioSnap for Fusion-io cards are expensive, requiring a full rebuild of the FTL. Techniques to checkpoint or build the FTL in the background are required to mitigate this problem.

- Any system built on flash must acknowledge the existence of foreground and background tasks and understand its impact on each other. Background tasks like segment cleaning and data scrubbing can interfere with the foreground performance. Rate-limiting background tasks is essential. Be it, host based or hardware FTL, a flash based device always has foreground and background tasks that compete with each other for access to data and metadata.

In ioSnap, we have explored a series of design choices that guarantee negligible impact to common case performance, while deferring infrequent tasks. ioSnap represents an extreme design point that has helped us understand the extent to which background tasks and snapshots interrelated. Unfortunately, in production systems, such large activation times are not practical (be it during backup to a slower media or during restore). Thus, ioSnap reveals performance trade-offs one must consider while designing snapshots for flash.

4 MODERN INTERFACES TO FLASH

Hard-disks, tape drives and cdroms exposed a simple block interface as they were either write-once or write-in-place media. Unfortunately, flash is a significantly different medium. Flash requires a log and an FTL to present a write-in-place interface. It is the presence of these two new entities that enable the creation of new, flash-aware and flash-enabled interfaces.

In this chapter, we present a case for rethinking the interface exposed by flash. The naive block interface limits flash's capabilities and impacts application performance and can prove detrimental to performance. Fortunately, the effort that went into presenting a traditional block interface can be leveraged to provide native interfaces to flash.

We describe properties of the FTL and the log that form the starting point for the new interfaces we build. The log represent a time-ordered sequence of data written by the application, while the FTL helps virtualize the address space. We leverage these two concepts to develop new APIs to natively virtualize address space and time.

We present a small set of new APIs, namely, range clone, move and merge that will help virtualize space and time. These APIs are primarily used to manipulate the mappings in the FTL. Cloning a range results in copying the logical address space to physical mappings to a new destination address. Moving a range is equivalent to cloning the range followed by trimming the source range. Finally, merging a set of ranges can help merge a vector of logical ranges into one destination range based on some policy.

With the help of these APIs, applications can express their intentions or requirements better. To demonstrate the power of these APIs, we modify existing file systems and applications, to use these APIs, to help perform their tasks better on flash. For example, a file copy is no longer a read of a file, followed by a set of writes. Instead, a file copy corresponds to cloning the source address range to the destination file's address range. Similarly, MySQL can benefit from atomic writes implemented using clones to avoid maintaining its own internal log.

The rest of the chapter is organized as follows. In Section 4.1, we first make a case for the need for newer interface. Next, in Section 4.2, we present the element of flash that act as the enablers for new interfaces. We go over the details of the high-level and low-level interfaces in Section 4.3 and Section 4.4 respectively. We briefly describe the implementation details in Section 4.5 and in Section 4.7, we present a detailed case study of applications leveraging the new interfaces. We finally conclude in Section 4.8.

4.1 THE NEED FOR NEWER INTERFACES

Standard block devices have been providing the read/write interface for a long time. Thus, any new device, irrespective of the medium, attempts to first deliver a block interface to allow immediate adoption. Flash is no exception to this norm having broken into the market first as a solid state device (SSD). SSDs primarily behave as block devices, with a similar form factor and requiring a standard SCSI or SATA port to connect. Fusion-io cards and recently other providers have switched to the PCIe bus to maximize the throughput the cards can deliver [40]. But predominantly, every NAND flash vendor still provides only the standard block interface.

Though the block device approach was the right approach to smooth the transition into the realm of non-volatile memory, it might not necessarily be the best in the future for several reasons.

Limits expressiveness and increases complexity of software

A simple read-write interface provided by the block device limits the extent to which the intent behind operations can be expressed. A simple high level operation translates to a complex sequence of operations. For example, a multi-block update becomes an update to the journal followed by an in-place update. In addition, the simplistic interface also imposes ordering requirements on operations involving data and metadata which complicates application design and can result in bugs [75]. Finally, to ensure correctness of operations in the event of crashes, applications (including file systems) are forced to employ complex recovery protocols and consistency checkers (fsck [17, 50], Aries [84]) to avoid inconsistencies created primarily by the simplistic nature of the block interface.

Limits performance

The presence of complex protocols above the block interface automatically impacts performance. File system (in data journaling mode) result in the data written to the journal as well as the final destination after the journal commit. Thus, the requirement of two writes puts an upper limit on performance. Another example related to the journal is the MySQL double write buffer. MySQL uses a double write buffer to write the committed data. Only upon completion, the committed transaction is written to the respective tables.

Limits device lifetime

In some cases, like double write and (data) journaling scenarios, the simple read-write interface can be attributed with reducing the flash device's lifetime. With all the performance advantages of flash comes the limited write lifetime. A flash device is calibrated to handle only a specific volume of writes in its lifetime (e.g., ioDrive2 can absorb x PB of data). Thus, a filesystem or application working on top of flash must be judicious with its data writes if it wants to extend the device's life. For example, by doubling the writes (due to a journal), the file system halves the device lifetime [95].

4.1.1 How SSDs limits Flash's capabilities

We now explore specific example of how the block device interface can limit traditional NAND flash's inherent capabilities.

Log is hidden

As described in Chapter 2, NAND flash implements a log to absorb writes to avoid performance and device endurance woes. Unfortunately, the block interface hides the log underneath. A log, by design, represents the sequence of writes that occurred on the device and thus, inherently represents the time order of updates. By leveraging the log, one can easily do away with the double writes to the journal and the data by marking a set of writes as a transaction (i.e., if even one of the writes in the transaction is missing, the transaction is discarded). In Chapter 3, we demonstrated the usefulness of the log in creating snapshots. Thus, the log is a powerful abstraction that can be leveraged to create many interesting applications.

Address space virtualization is hidden

NAND flash typically employs an FTL to virtualize the address space in order to help hide the log underneath. The presence of the virtualization layer is hidden by the block interface that only exposes a large array of sequential addresses. Applications can take advantage of the existence of a virtualization layer to implement a lot of interesting features as we will present in the later sections.

Application interference

Applications or file systems may inadvertently interfere with the flash device operations. For example, a file system may attempt to create snapshots in a flash unaware manner and may generate unnecessary writes to the log to update reference counts or copy snapshotted data. Modern file systems like btrfs [6], WAFL [51] are based on copy-on-write and thus, they build a log. Unfortunately a log on top of a log is not always the smartest thing to do. The garbage collection operation occurs at two levels and when left uncoordinated, it can result in suboptimal performance.

4.1.1.1 Impact on Application Performance

Applications (or file system), running oblivious of flash underneath the block device, can suffer from suboptimal performance. We describe some scenarios in this section.

File system block allocation policies

Most file system block allocation policies have been designed with a disk in mind. They attempt to group blocks of a file together, files in a directory to the same cylinder group etc [79]. Unfortunately, with flash, all these optimizations are unnecessary on top of a log which only appends data. Btrfs [6] attempts to be flash-aware by always allocating blocks from the head of its log.

Logging for consistency

File systems, unaware of being run on top of flash and its capabilities, write to a journal. As we observed before (in data journaling mode), a journal creates a log on top of log and is both detrimental to the performance and the life of the device. Similarly, MySQL suffers from poor performance when using the double write buffer.

Key-Value stores on flash

Typical key-value stores implemented on top of SSDs, require complex garbage collection policies to help cleanup their data [73]. This is another good case of a log on top of a log, with the two levels of garbage collection interfering with each other and may result in suboptimal performance.

Thus, a simple block interface not only limits the capabilities of flash, it also can be detrimental to application performance and device life. In order to fully realize the flash's potential, we need to explore newer, native interfaces. The rest of the chapter is organized as follows.

First, we describe various axes of control enabled by flash, namely address space, time, persistence and mutability. Next, we present a set of high level APIs which are exposed by our system using the axes we described. This is followed by a more detailed description of the low level interface our system implement and an evaluation of the costs of our system. Next, we present some interesting cases which demonstrate the usefulness of our interfaces before concluding.

4.2 AXES OF CONTROL

As we observed in the previous section, building applications in a flash-unaware manner is detrimental to the application or file system. Even ignoring performance impact of a naive application, the effect on the device wear is significant is some cases (for example, MySQL double writes reduce device lifetime by half). Thus it becomes critical to explore newer interfaces that enable applications to interact natively with flash. In this section, we explore some of the axes of control enabled by flash around which newer interfaces can be designed. We also introduce the high level interfaces we propose to expose to consumers of flash and then describe the requirements of such an interface.

4.2.1 (Address) Space

The address space exposed by the block device is the range of addressable blocks available on that device. Typical disk based devices have a one-on-one mapping between the address space exposed to the OS and the actually physical addresses used internally to access data. This is enabled due to the fact that disks allow in-place overwrites and a one-to-one mapping is always feasible. Unfortunately, as we observed in Chapter 4.1, NAND flash has some interesting characteristics that make in-place overwrites both slow and bad for device lifetime. Thus, an FTL was introduced to handle the mapping between the logical or user-visible address and physical or internal address.

The presence of the indirection layer (or FTL) opens up a whole range of interesting opportunities. The simplest of these are a sparse, seemingly infinite logical address space mapping to a much smaller physical address space. Flash devices, when used as a block cache, can expose the address space of the backing device, while having only a fraction of the real device capacity [39, 110]. Other applications like a key-value store [42] can also leverage a sparse address space to create a low overhead store that randomizes keys over a large (and sparse) address space. Similarly, DirectFS [38] also leverages the sparse address space to keep block allocation simple. Thus the presence of the FTL has enabled applications to perform their tasks more efficiently.

The presence of an FTL also enables some address space sharing capabilities. The FTL maps a logical address to a physical address. Thus, it also naturally allows more than one logical address to point to the same physical address. Sharing data among entities thus becomes simple. Moving data from one address range to another also becomes simple: remap from the source range to the destination range. Thus, the crux of the address sharing capability is the ability to modify the logical address space without making any modifications in the physical address space.

4.2.2 Time

Data stored in most storages devices is modified over time (intentionally or unintentionally). With in-place overwrite devices like hard-disks, the trail of changes is immediately lost and so is the file history. Systems like ext3cow [96], Btrfs [6], Apple Time Machine [13] have been designed to provide the notion of time on top of write-in-place file systems by playing a lot of tricks within the file system metadata to make it safely store versions of the file.

Flash on the other hand leverages a log and as we observed logs by definition represents the set of all changes that resulted in the current state of the system. Thus, by reading a log up to a certain point, we can retrieve the state of the system at that point. The log of course has to cleanup old data to recover space, but as we described in Chapter 3, we can easily modify the driver to provide point-in-time snapshots of the device.

Thus, the existence of the log creates a time sequence of operations and enables creation of applications that can natively request the state of the system at any point in time (in the past!).

4.2.3 Persistence and Immutability

As we described earlier, the state of the log is sum of the changes represented on the log. Data found on the log, can thus be given other attributes like crash persistence and mutability.

4.2.3.1 Persistence

Crash persistence determines whether a piece of data will be considered valid after a crash. For example, a database transaction engine may be writing out new data as part of a transaction update and may crash before committing. The typical database recovery engine [84] would replay the redo or undo logs and figure out the updates that must be retained and the ones that need to be thrown out.

To make crash recovery simpler, the transaction engine can mark uncommitted data as non-persistent. Thus, the underlying storage engine can discard non-persistent data after a crash whereby simplifying the recovery process of the database.

4.2.3.2 Immutability

Typical data management system (like file systems) provide the ability to mark entities as immutable through access control lists. Immutability is a useful attribute for systems that take periodic backups or share vast amount of data or programs among different users of the system. Unfortunately, these ACLs are only useful when they are enforced. When an application bypasses the ACLs, the standard disk driver has no knowledge of what data can and cannot be modified.

The presence of the indirection layer in the case of flash provides an elegant solution. No application can bypass the FTL and thus evaluating immutability criterion becomes simple.

4.3 HIGH LEVEL INTERFACES

Given the various axes of control enabled by NAND flash-based devices, we now present some interfaces that help expose new functionality around these axes. We first illustrate an example, a sequence of operations bringing in all the axes of control we listed earlier. With the example as the backdrop, we describe a set of interfaces that will help provide a comprehensive API to control these axes.

A Virtual Address Space (VAS) represents one or more contiguous address ranges belonging to the overall address space exposed by the device. Figure 4.1. illustrates an example showing the operations one can perform with a VAS. We start out with a virtual address space, lets call it VAS1. It is possible to talk about this address space as existing in different points in time (frozen in time). For example. VAS1(at t1) (VAS1 as it was at time t1) was frozen and hence available at any time in the future, whereas VAS1(at t2) was not frozen and thus not available at the current time. These versions are by definition immutable since they are frozen in time 4.1(a).

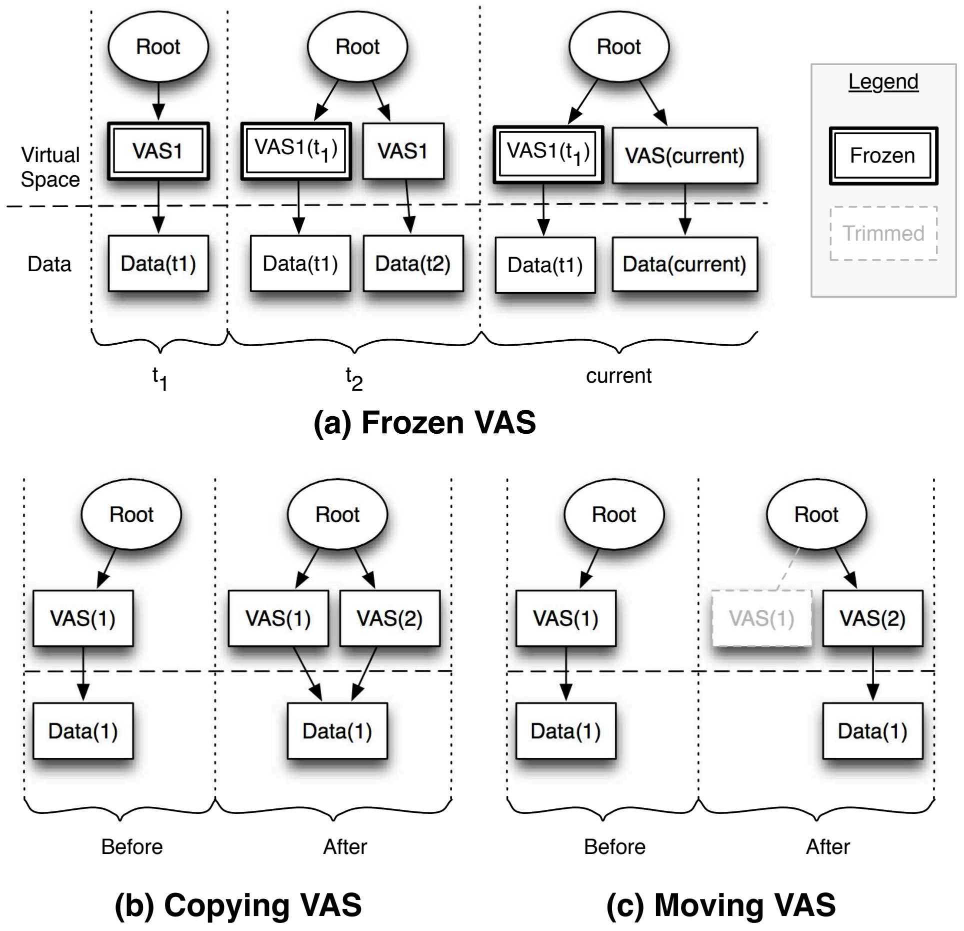

Figure 4.1: **Virtual Address Space (VAS) Operations.** *The figure above illustrates the various address space operations we support. We represent the state of the device as a tree that stores the various mappings between the virtual addresses and the physical addresses. Virtual Address Space (VAS) are contiguous regions of the address space. VAS(t1) in (a) represents the state of the VAS at time t1 and VAS(t1) is frozen. A frozen VAS is available at any point in the future until it is explicitly deleted. (b) and (c) depict two scenarios where data present in VAS1 can be copied or reassigned to another VAS2.*

Multiple virtual address spaces may also point to the same physical address. For example, VAS1 and VAS2 share the same data after VAS1 was copied to VAS2 4.1(b).

Finally, it is also possible to reassign data to a different virtual address. For example, VAS1 could move to VAS2 4.1(c).

Given this model, VAS(t1) can never be changed (since it is VAS1 at time t1, which will never come again). However, one could have VAS1(t1) share physical blocks (copy) with a new VAS, VAS2 and VAS2 is now mutable.

We now describe the interfaces that help provide the ability to control address space, time and mutability.

4.3.1 Virtualizing Time

The following interfaces help virtualize time:

4.3.1.1 Freeze VAS in Time

Freezing a VAS is equivalent to taking a point-in-time snapshot of the content of the VAS. The frozen VAS is persistently available henceforth.

4.3.1.2 Activate Frozen VAS

Once a VAS has been frozen, it may be activated at any time in the future. After a VAS is frozen, the source of the frozen VAS may be modified and thus activation of a VAS frozen in time requires reconstructing the state of the VAS at that point time.

4.3.2 Virtualizing Space

The following interfaces help virtualize space:

4.3.2.1 Mark VAS read-only / writable

A VAS can be made read-only or writable by invoking this API. With the help of this API, we can mark regions of the logical address space immutable (for example, a snapshot of a volume). A read-only VAS is different from a frozen VAS: frozen VAS' are available across time and cannot be modified directly. A read-only VAS ensures the mapping and the data both cannot be modified, unless explicitly made writable.

4.3.2.2 Copy Source VAS to Destination VAS

A contents (i.e. mappings) of a VAS can be copied to another VAS, which results in both VAS sharing the physical blocks. Figure 4.1(b) illustrates an example where VAS1 was copied to VAS2 and both ended up pointing to the physical address PAS1. It is important to note that the source VAS may be a VAS that was frozen in time, which makes it similar to the Activate Frozen VAS API.

4.3.2.3 Move Source VAS to Destination VAS

Move remaps the physical address blocks pointed to by the source to the destination VAS. Figure 4.1(c) illustrates an example.

4.3.2.4 Delete a VAS

Deleting a VAS provides the ability to discard the contents found in a VAS. The VAS may also be one that was frozen in the time.

4.4 LOWER LEVEL INTERFACES

In this section, we describe the low level interfaces we design to expose the interfaces we defined in Section 4.2.

4.4.1 Lower Level Interfaces

We expose three basic primitives that can be leveraged by a higher level system to achieve complex address space and time virtualization.

The lower level interfaces are

4.4.1.1 Range Clone

A typical range clone operation would look like the following:

range_clone(source vector, destination vector)

The source vector is a set of address ranges that must be copied to a destination range. Upon successful completion of the clone operation, the source and destination ranges both point to the same physical addresses.

4.4.1.2 Range Move

The range move operation is used to move data in a one or more address ranges to a destination address range. The mappings from the source vector are moved to the destination vector. A typical range move operation would look like the following:

range_move(source vector, destination vector)

4.4.1.3 Range Merge

Range merge is a primitive that can be used to merge the data found in a vector of address ranges (based on some merge policy) and store the result in the destination range. The primary use case for the range merge operation is during version control. A frozen address range 4.1(a) corresponding to say a file, may be cloned to two different locations and operated upon by two different processes. Upon completion, the two destination ranges may be merged into one destination range to reflect the operations by both processes, based on some policy that can detect and resolve conflicts.

A typical range merge operation looks like the following.

range_merge(source vector 1, source vector 2 ..., source vector n, destination vector, merge policy)

The merge policy is also tasked with the job of figuring out if there exists a conflict in the source vectors. A simple merge policy may look for conflicts and fail when one is detected. Another merge policy may blindly pick the first vector when a conflict is found.

4.4.1.4 Range Attribute

The last interface, range_attribute is used to specify the attributes of ranges like persistence and immutability. The API for setting the range attribute takes a address range vector and the attribute to set.

range_attribute(range vector, PERSISTENT |
NON PERSISTENT | IMMUTABLE | MUTABLE)

4.4.2 Implementing High-level Interfaces

The high level interfaces are meant to be a library on top of the low-level interfaces and in this section, we illustrate the implementation of the high-level interfaces using the low-level interfaces.

4.4.2.1 Time

We leverage the lower level interfaces to virtualize time as follows:
Freeze VAS in Time

Freezing a VAS requires cloning the given source address vector into a reserved address range that is immutable by default. The immutable address range is never reallocated unless explicitly released through a

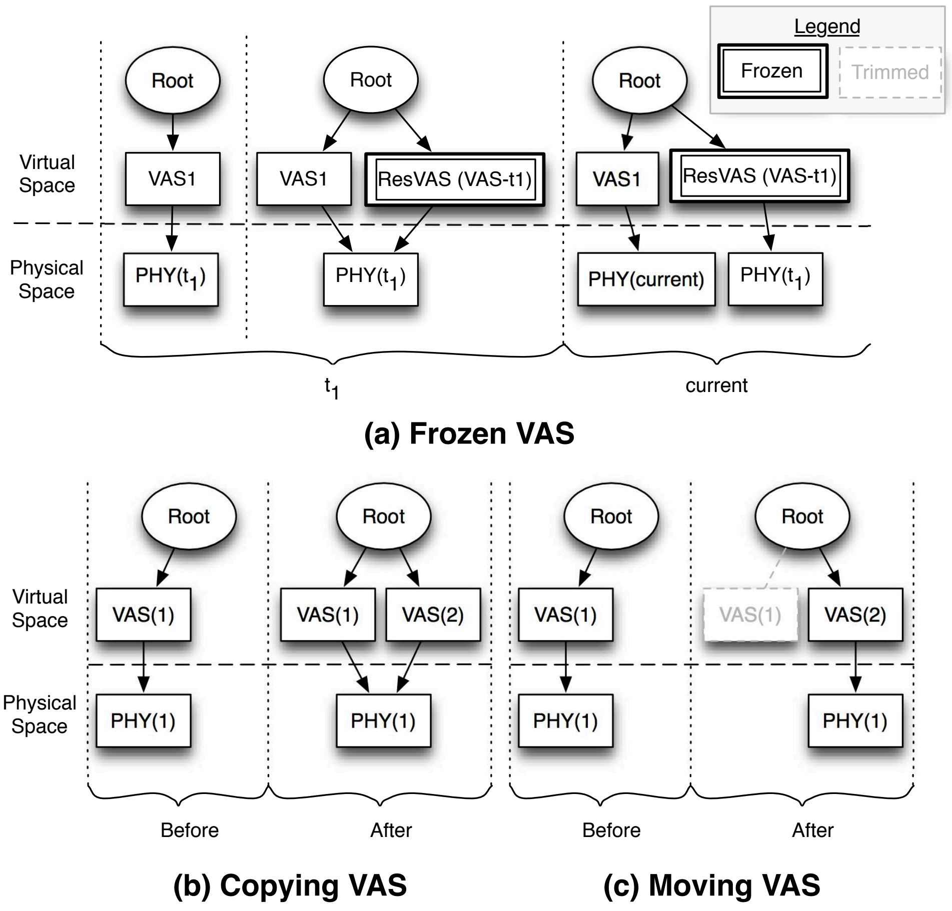

Figure 4.2: **Implementation of VAS operations.** *The figure above illustrates how the VAS operations describted in Section 4.2 can be implemented using the lower level range operations. Fig a illustrates freezing a VAS. Freezing a VAS involves cloning the range to a reserved range. In our example, we clone the range VAS1(at t1) to ResVAS. Thus the reserved range is available in the future storing the contents of VAS1 at time t1. Fig b and Fig c illustrate the use of range clone and move to share and reassign data across VAS.*

Delete VAS call. The API returns a time that acts as the handle to retrieve a frozen VAS. Figure 4.2(a) illustrates an example.

Algorithm 1: freeze_vas(source address)

```
    Input: source address: address to freeze
    Output: time: current logical time
    range_clone (source address, reserved address);
    store_map (source address, time, reserved address);
    return time;
```

Algorithm 2: store_map(data)

```
    Input: key(addr1, time) : key to store in hash
    Input: value(addr2) : value corresponding to key
    write_log (data);
    hash ((addr1, time)) = addr2;
    return;
```

Algorithm 3: retrieve_map(key)

```
    Input: key(addr1, time) : key to search hash
    Output: value(addr2) : value corresponding to key
    return hash ((addr1, time));
```

Activate Frozen VAS

Activating a frozen VAS requires cloning the reserved address range corresponding to the frozen VAS into a new address range specified by the user.

Algorithm 4: activate_frozen_vas(source address, time, destination address)

```
    Input: source address: address to freeze
    Input: time: current logical time
    Input: destination address: address to load frozen VAS
    reserved address = retrieve_map (source address, time);
    range_clone (reserved address, destination address);
```

4.4.2.2 Space

We leverage the lower level interfaces to perform address space virtualization as follows:

Mark VAS read-only / writable

Modifying the mutability of a VAS is a simple operation. We have to directly invoke the API to modify the range attribute as follows.

range_attribute(address range, IMMUTABLE | MUTABLE)

Copy Source VAS to Destination VAS

Copying one VAS to another is implemented in one of two ways depending on the nature of the source range.

Example 1. Live ranges

Live ranges are address ranges from a active volume. Copying such ranges are simple, involving a call to the range clone API. An example is illustrated in Figure 4.2(b).

range_clone(source address, destination address)

The contents of a frozen range can be modified by copying the range to a new destination range using the following method:

activate_frozen_vas(source address, time, destination address)

Move Source VAS to Destination VAS

A VAS can be physically moved to a new VAS by invoking the range move API. An example is illustrated in Figure 4.2(c).

range_move(source address, destination address)

Delete a VAS

Deleting a VAS involves issuing a block trim on the address range [61].

4.5 IMPLEMENTATION DETAILS

In this section, we present the implementation of the Clones subsystem on top of the existing Fusion-io VSL driver. First, we describe the need for an additional layer of indirection and how this simplifies garbage collection and address space sharing. Next, we describe the approach adopted to persist the addressing layer. As observed with the FTL in Section 2.2.4, the second layer of indirection also needs a crash recovery mechanism and we discuss some of the details. Finally, we put together the Clones subsystem with the VSL driver to achieve the Range operations listed in Section 4.4.1.

4.5.1 Need for Two-Level Indirection

In the presence of an FTL, it would seem obvious how range operations could be implemented. The FTL is a b-tree and a range clone or a move only requires copying or moving the logical to physical address mapping to the new logical range.

Unfortunately, the garbage collector complicates matters. The garbage collector can, asynchronously, move physical blocks around and as part of the move completion, it updates the FTL to indicate the new location. In the presence of clones, more than one logical address may be pointing to the block that got moved. Thus, the garbage collector has to ensure it has updated all the forward pointers that may be pointing to a physical block that was moved. The garbage collector, which originally had predictable performance, now has to contend with a lot of variability. Figure 4.3 illustrates an example. As the example illustrates, in the presence of clones, the garbage collector may have to manipulate multiple mappings resulting in performance variability.

A simple solution to avoid such variability is by introducing a second layer of indirection. There are two approaches to use a second level of

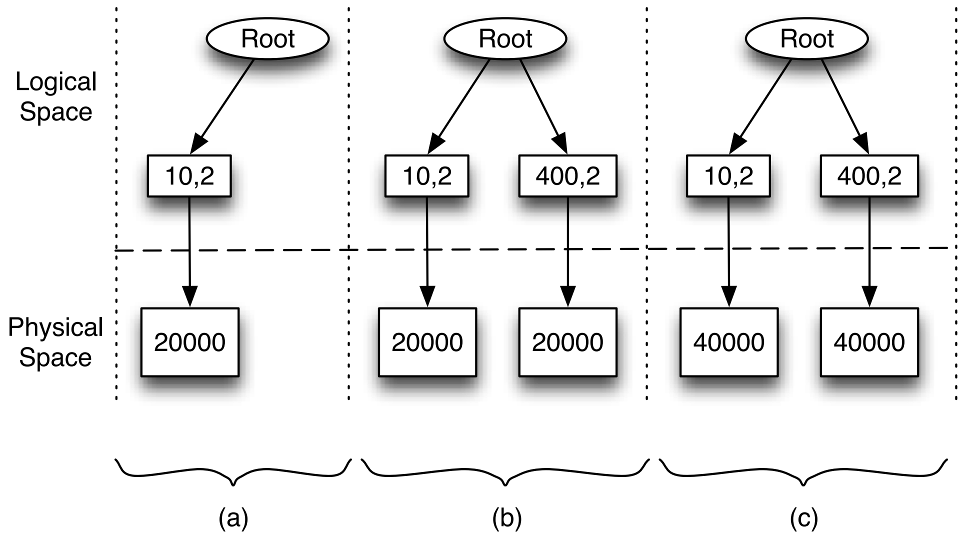

Figure 4.3: **Garbage collection with Clones.** *With the range clone operation, we consistently land in a situation where two or more logical addresses point to a physical address. An address range, reresented by the tuple (x,y), corresponds to a set of addresses starting at address x, spanning up to address x+y. In the figure above, logical address range (10,2) was originally pointing to physical address 20000 (a). Upon cloning the range (10,2) to (400,2), both ranges now point to the same physical address (b). When the garbage collector attempts to move physical block 20000 to new physical address 40000, it has to update two entries in the FTL (10,2) and (400,2). This process introduces a lot of unpredictability to garbage collector performance.*

indirection and both solves our garbage collector problem.

The first approach uses a parallel layer of indirection. The primary FTL stores all the indirections and the second layer is only introduced in the presence of range operations. Figure 4.4 illustrates the example. In the example, we clone a range starting at logical address 10 and length 2 (represented as 10,2) at physical address 20000 to destination address 400. Instead of inserting a new entry in the primary FTL mapping logical (400,2) to physical 20000, we introduce the second level. We allocate a region of length 2 from the second level (say 100000,2).

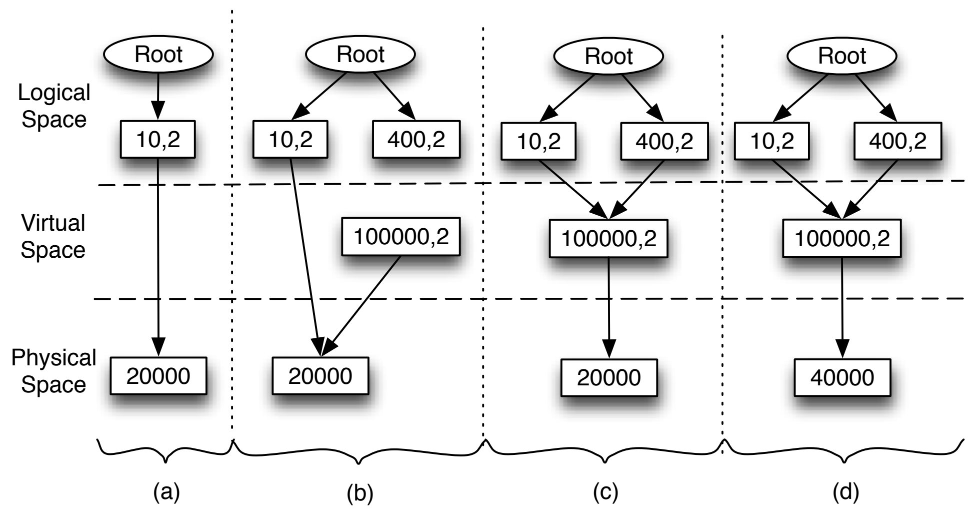

Figure 4.4: **Parallel Two Level Addressing.** *The figure above illustrates the use of two level addressing to address the garbage collection issues. We use two level addressing with the second level only used in the presence of clones. An address range, reresented by the tuple (x,y), corresponds to a set of addresses starting at address x, spanning up to address x+y. As Fig(a) shows, in the absence of clones, the primary FTL stores the mappings between Logical and Physical space. The moment we clones range (10,2) to (400,2) we introduce the second level (b). The first step involves allocating a new virtual range (100000,2) and install a new mapping between (100000,2) and physical address 20000. The next step (c) involves updating the logical address ranges (10,2) and (400,2) to point to virtual address range (100000,2). With the virtual address indirection in place, moving physical block 20000 to 40000 only involves updating the virtual or secondary mapping(Fig d).*

Next, we insert the original mapping into the second level (i.e., logical 100000,2 to physical 20000). The final step involves updating the primary FTL to point to the secondary FTL (i.e., logical 10,2 and 400,2 maps to secondary 100000). The advantage of this approach during garbage collection is that only one location would have to be updated when physical block 20001 is moved. The primary level stays oblivious of this specific block move. It is important to note that the second level of addresses are virtual addresses that are internally managed by the Clones subsystem to simplify garbage collection and never exposed to the application.

The second approach of leveraging indirection layers the second level below the primary FTL. This approach is conceptually similar to the first approach, but with a significant difference. The two levels of indirection are used during every translation instead of just during clone accesses. Figure 4.5 illustrates an example. The primary level (ie the level on top) stores mappings between the logical address seen by the application and an intermediate virtual address presented to the secondary FTL (ie the level on the bottom). The secondary level (or FTL) stores the mappings between intermediate virtual addresses and physical addresses. The secondary level (or intermediate virtual address) is an address space that is internally managed by the clone layer and never exposed to the application. The same example as above would work the following way. For example, lets say the range (10,2) maps to virtual range (100000,2), which in turn maps to physical address 20000. When the range (10,2) is cloned to (400,2), we only insert a new mapping in the primary layer: (400,2) maps to (100000,2). Clearly, when physical block are moved, only the secondary level (at the bottom) needs to be updated. For example, when physical block 20001 is moved, only intermediate virtual address mapping 100001 needs to be updated and the primary level stays oblivious.

We chose the second approach for the Clones prototype for the fol-

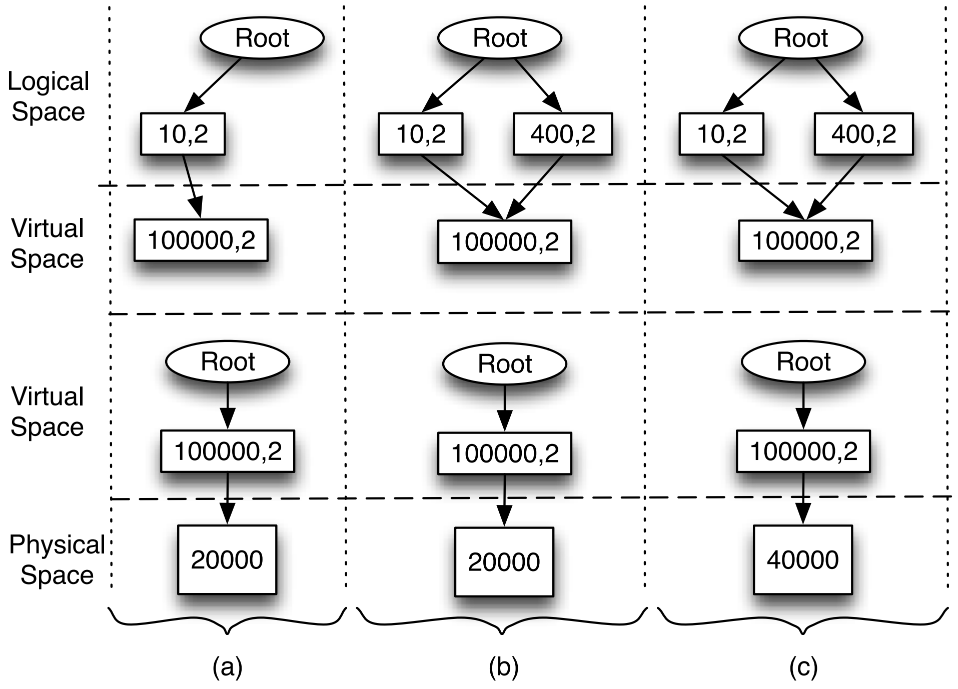

Figure 4.5: **Layered Two Level Addresses.** *The figure above illustrates the use of two level addressing to address the garbage collection issues. In particular, we use the two levels in a layered mode with both layers involved in all lookups. An address range, reresented by the tuple (x,y), corresponds to a set of addresses starting at address x, spanning up to address x+y. As Fig(a) shows, even in the absence of clones, the primary FTL stores the mappings between Logical and Virtual space (e.g. range 10,2 is mapped to virtual range 100000,2). The tree above shows the logical to virtual mappings while the tree below shows the virtual to physical mappings. Cloning a range only involves copying the virtual address mappings. Fig b shows cloning of range (10,2) to (400,2). As before, moving physical block 20000 to 40000 only involves updating the virtual to physical mapping, thus constraining the number of mappings to be updated.*

lowing reasons. The second approach simplified the overall implementation by allowing us to stay outside the FTL for all practical purposes. Our initial experience with implementing a second, parallel layer was not very positive. We were constantly forced to deal with the internals of a very complex FTL while trying to understand the new features were were adding. Thus, we decided to chose layering and understand the usefulness of the interfaces we were adding rather than retrofit new features into an already complex FTL. The obvious tradeoff in this approach is the loss in performance. When an additional translation is used in every lookup, the read and write performance is bound to drop and we quantify the overheads in our evaluation.

4.5.2 Operations on a Two-Level System

In the presence of a second level of indirection, the operations performed on the device take a modified path. We describe the IO path and the implementation of the range operations in this section.

Regular IO

The regular IO (read and write) path has to deal with the presence of the second level of indirection. The read and write request for logical addresses are first translated to virtual addresses. A write may require allocation of new virtual addresses. The request, now containing virtual addresses are sent to the lower level FTL that handles the physical operations. We employ two types of allocators to handle issuing and freeing of virtual addresses: a simple free list and a extent-based buddy allocator. The virtual address is now a resource that must be managed by the clone system and needs a garbage collector (we have not implemented garbage collection, but we describe the high level overview in Section 6.1.1).

Range Operations on Two-Level System

Range operations exclusively work with the top-level (or primary)

FTL. A range clone operation copies the top level mappings (logical to virtual) to the destination. Similarly, range moves require a remap of logical to virtual mappings. To persist a range operation, we write a note to the underlying medium stating the logical addresses and the virtual addresses involved.

4.5.3 Persisting Second Level

The use of a three level addressing scheme (application visible logical address, intermediate clone-only virtual address and physical NAND address) simplifies the garbage collection problem, but introduces another problem. The physical media only records the logical addresses issued by the lower level i.e., in our case the (invisible) virtual addresses from the intermediate layer. Thus, without explicitly persisting the mapping between the application level logical address and the intermediate virtual address, a crash-proof solution would be impossible.

The simplest solution to this problem is to persist all the layers of address on the media in out-of-band (OOB) area. Unfortunately, the block header is of limited size and it cannot accommodate more information. So the next obvious thing to do is to persist the mappings by writing a note with the block indicating the logical to virtual mappings (the block already has the virtual address). Unfortunately, persisting two blocks for each single block write is a performance hog (both space and time). Instead, we chose to group commit these address translation notes. A write operation is not complete without its translation note persisted. Thus, writes are suspended until their corresponding note is persisted. We evaluate the impact of group commits on the latency observed by the application and the overall impact on performance. In the absence of a note, all the virtual addresses tracked by the missing note are considered invalid and discarded.

4.6 EVALUATION OF CLONES

The goal of our evaluation is to demonstrate the practical feasibility of our system by measuring the overheads imposed by the clone layer on regular operations as well as the cost of performing clone operations. The clones subsystem spans over 2400 lines of new code and close to 10000 lines of unmodified FTL code, which we share with the native FTL driver. All the tests were performed on a 16 core Intel Xeon 2.4 GHz processor with 48 GB of RAM, running Linux 3.4.12 with a 2.4 TB Fusion-io IODrive card.

4.6.1 Regular Operations

In the presence of the clones subsystem, every block request incurs overheads imposed by the remapping we have introduced. In this section, we evaluate the pure cost of the clones layer i.e., overheads incurred on regular block requests in the absence of clones.

	Range Operations / sec
Without notes	388 K
With notes & group commit	207 K

Table 4.1: **Performance of Range Operations.** *The table above illustrates the performance of range clones with group commits. Group commits of range operations accumulate notes and persist the note only when they exceed a single sector (4KB). Thus, each range note can now accommodate upto 170 range move operations (size of one entry is 24 bytes). We perform 1 million 1 sector (4k) range moves. In the absense of notes (i.e., non-persistent range operations), we hit our peak performance of 388K range ops/sec, while with group coummits we hit a peak performance of 207K range ops/sec.*

We performed a large number of sequential and random 4KB reads and writes and compared the clone device performance with the native

	Seq. Write (usec)	Seq. Read (usec)	Rand. Write (usec)	Rand. Read (usecs)
Raw Device	44.3	112.34	44.4	114.78
Clone Device (no active clones)	45.5	115.55	45.65	117.09

Table 4.2: **Performance of Regular Operations.** *The table above present the results of a series of micro-benchmarks to understand the impact of the clones shim-driver on regular read and write operations. We performed 1 millions sequential (or random) read (or write) operations of 4KB each on the clone block devic (using direct io). No clones or moves were issued during this test. As we can observe, the overheads imposed by the clone block device at most 2-3%. Being a prototype implementation, we have kept the clones layer independent of the underlying device driver: when we merge these two layers, this overhead could be reduced further.*

	Seq. Write (usec)	Rand. Write (usec)
Clone Device (no active clones)	45.5	45.65
Clone Device (with clones)	49.25	50.36

Table 4.3: **Performance of Regular Operations with Clones.** *The table above illustrates the cost of breaking clones. We performed 1 millions sequential writes of 4KB each on the clone block device (using direct io). Next we performed a range clone over the 1 million sectors and then started writing to the destination. The overheads seen by the write operations are of the order of atmost 10%. Though a 10% overhead is seemingly large, it is a one time cost incurred the first time a clone is broken.*

block device. In all these test, we have formatted the drives to work with 4KB block sizes and we performed a large number of reads and writes. Table 4.2 shows the performance numbers we observed in these tests. As we can observe, the performance impact of the clone layer is $\approx$ 2-3%, which is understandable since we implemented the clone system as a separate layer in our prototype.

4.6.2 Range Operations

Range operations involve updating the upper-level translation layer and also writing a note to the log to persist the change. In this section, we evaluate the cost of performing these operations.

The first workload we run simply issues a range clone operation of a single randomly picked sector and forces a note to be flushed to the log to persist the operation. The size of each note is 4KB irrespective of the number of sectors moved. The time spent performing each operation is around 53 μsecs on an average. We can attribute 50μsecs to the cost of writing a note and around 3 μsecs are spent performing the manipulation on the mapping layer. So, the worst case performance comes to about 19K range operations per second. Another thing to note here is the cost of performing a range clone is equivalent to the cost of a range move as both require the same number of passes through the mapping layer.

The next workload attempts to understand the best case performance. The reason behind the poor performance was clearly the need to write a note for each range operation. Instead we adopt a group commit policy: accumulate range operations and write a note only when we run out space in a note(4KB). Thus, each range note can now accommodate up to 170 range operations (size of one entry is 24 bytes). Table 4.1 presents the results of our experiments. We perform 1 million 1 sector (4k) range moves. In the absence of notes (i.e., non-persistent range operations), we hit a peak performance of 388K range ops/sec, while with group commits we hit a peak performance of 207K range ops/sec. Group commits can help improve performance of the range operations by at least 10x.

4.6.3 Cost of Breaking Clones

Range clones are implemented with the help of mappings that translate two different logical addresses to the same virtual addresses. Thus, when a write is issued to one of these ranges, then we are required to allocate a new virtual address to remap the write. Table 4.3 illustrates the overheads involved during clone breakage. We performed 1 million sequential write operations of 4KB each on the clone block device (using direct io). Next we performed a range clone over the 1 million sectors and then started writing to the destination. The overheads seen by the writes are of the order of at most 10%. We can attribute the overhead primarily to the additional time spent inserting new entries into the top-level mapping layer. Though a 10% overhead is seemingly large, it is a one time cost incurred the first time a clone is broken.

4.6.4 Summary of Results

The new interfaces exposed by the clones subsystem imposes minimal overhead on regular operations and the one-time overhead when the clone is broken can be tolerated. The range operations provide sufficiently high throughput to make these realistic. Thus, the current state of the clones infrastructure makes it practical to explore realistic use cases.

4.7 CASE STUDIES WITH CLONES

Clones are a powerful primitive that allows a large variety of applications to accomplish their tasks in a simpler and often more efficient manner. In this section, we go into some of the interesting use cases of clones and illustrate how applications can benefit from using clones.

4.7.1 Direct File System (DirectFS)

To show the power of clones, we take DirectFS and implement several features with minimal code changes and implementation effort [38, 62]. DirectFS is a lightweight file system designed in a flash-aware manner to leverage the features of the Fusion-io FTL to achieve very high performance.

DirectFS leverages the Fusion-io FTL in a couple of interesting ways to deliver its performance. The FTL has the capability to expose a sparse address space that is significantly (several order of magnitude) larger than the space available in the underlying device. By exposing a very large address space, file allocations don't have to follow the complex mechanisms designed for the hard disk (like direct, indirect blocks). Instead, DirectFS adopts an extent-based scheme, allocating blocks from a series of allocation groups (increase in size) as and when needed by the file. Since all logical addresses go through one level of translation, it nullifies the need for block allocation to understand the underlying device geometry. The use of simple extent based allocation makes DirectFS very lean and fast.

Secondly, DirectFS also leverages the Atomic Writes [92] feature present in the Fusion-io FTL. With the help of Atomic Writes, the Fusion-io FTL can guarantee the persistence of a vector of writes. This simplifies DirectFS by doing away with journaling (for consistency). Every transaction persisted by DirectFS gets written as an Atomic Write, thereby either persisting the full transaction or none of the individual components.

4.7.1.1 File-level Snapshots

File-level snapshots are a useful feature in modern file systems and only a handful of file systems support it [6, 119]. File-level snapshots enables applications to checkpoint individual file state (i.e., contents) that they care about over different periods of time. Unfortunately, not all file system support such a feature. For example, ext4 [78] and XFS [123] do not support file-level snapshots. The reason being there is a significant amount of data structure restructuring and implementation effort required to support per-file snapshots within the file system.

Clones enable file systems to support file-level snapshots with no changes to their internal data structures and also requires minimal implementation effort. Snashotting individual files becomes trivial with clones, as one has to only clone the block mappings of the source file and save it as part of another (hidden) file [51]. We modified DirectFS to support file-level snapshots. We added a simple file snapshot ioctl. The snapshot ioctl on invocation performs the following operations. First, it flushes the dirty pages of the source file that needs to be snapshotted. Second, it creates a new (snapshot) file within the same directory as the source file. Finally, it clones the address space mappings of the source file to newly created snapshot file using the range clone API described above.

We implemented file-level snapshots in DirectFS by implementing a simple ioctl within the file system. The overall implementation requires 20 lines of code that essentially invokes the underlying range clone operation. A file snapshot corresponds to a single 4KB range clone note on the underlying media irrespective of the size of the original file.

4.7.1.2 Copying Files Via Clones

The copy operation in glibc is inefficient especially when on top of a copy-on-write type of medium like flash. The cp coreutil manually copies all the blocks present in the source file to the destination file. We modified the cp program to utilize the clones features available in DirectFS. Copying a file is similar to creating a snapshot (i.e., a copy-on-write clone of the original file). We modified the cp program to invoke the DirectFS file snapshot ioctl. The destination file is created if not found and the subsequent call to the range clone API copies the address mappings from the source file to the destination file.

4.7.1.3 DirectFS Address Space Fragmentation

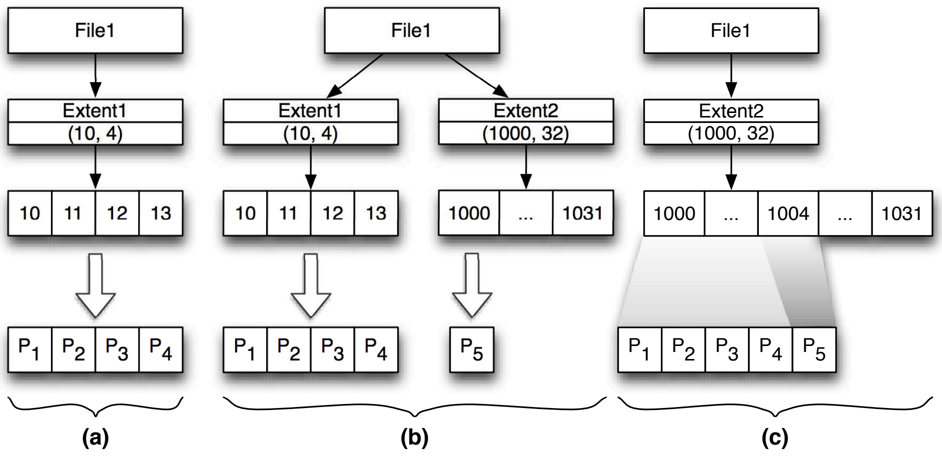

Figure 4.6: **Address Space Fragmentation in DirectFS.** *The figure above illustrates address space fragmentation in DirectFS when a file grows larger. Initially, File1 is allocated 4 blocks (address 10 to 13)(Fig a). When File1 grows larger, a new extent of a larger size (32 blocks) from address 1000 to 1031. Now, DirectFS has to store two extents to store the mappings from address 10 to 13 and 1000 to 1031 (b). With the help of a range move, we can move the original range 10-13 to 1000-1003 and store the new data in address 1004 (c).*

DirectFS relies on keeping file system metadata minimal to ensure high performance. Unfortunately, as file size grows, DirectFS also runs into an address space fragmentation issue as seen by other file systems. When a file exceeds its allocated size, a new address range from a larger extent group is appended to the file and new content are written to this extent. Thus, DirectFS has to keep a list of all extents that has been allocated to each file.

To avoid maintaining an extent list and always keep the file correspond to a single extent, we leverage range moves. The moment a file is allocated a larger range, we can issue a range move from the smaller range to the larger range. Figure 4.6 illustrates an example. We can observe that when file File1 grows beyond its allocated size (4 blocks), we allocate a larger extent range (32 blocks) and next issue a range move from the original range (10 to 13) to the new destination range (1000 to 1031).

4.7.1.4 Deduplication

Deduplication is the methodology of getting rid of duplicate data present in the data and making the files or volumes point to a single copy of that specific unit of data. We illustrated how file copies and snapshots can be accomplished with through clones in the previous sections. Deduplication is logically the reverse of those two approaches: instead of the user explicitly stating the need to copy or snapshot a file, we have to detect the existence of copies and internally mark them as clones of each other.

In order to illustrate the possibility of deduplication through clones, we built a simple user-level prototype that detects the existence of file-level clones i.e., are two files copies of each other. We periodically scan the DirectFS file system and create a dictionary of "md5" hashes we have observed during the course of a scan and the file's last modification time. When we detect a copy i.e., hashes match, then we issue a DEDUP file ioctl to the DirectFS file system. Before marking the files as clones of each other, the file system has to ensure neither the source or destination is in the process of being modified by other processes. We do this by checking for existence of a dirty inode and check if the modification time observed at the user-level has been modified now. If both inodes clear these requirements, we lock them and issue a clone ioctl to the lower layers for the ranges corresponding to these two inodes.

4.7.1.5 Checkpointed mmap

"mmap" is a commonly used construct that allows contents of files to be accessed as pages of memory through standard load and store operations instead of the read/write interface provided by the file system. To go with the mmap operation, file systems also provide the msync system call. msync allows the file system to flush all the dirty pages to the persistent store. A combination of mmap and msync can be used to achieve an effect similar to a set of read/write operations.

One of the primary concerns while using mmap is the inability to checkpoint the state of the file at any given time. Operations are performed in memory and an msync is issued when the state has to be saved. But, the state of file after an msync represents the current in-memory state and the last saved state is lost. Moreover, if the file system were to crash during the course of an msync, the file is left in an inconsistent state.

To avoid these issues, we propose a simple solution within DirectFS. We add the ability to checkpoint the state of an mmap-ed file during every call to msync. Checkpointing a file is equivalent to creating a snapshot of the file (as described above). The moment a file is mmap-ed, the file is checkpointed (or cloned). This clone saves the state of the file before any changes are done to it. Next, when an msync is issued, we create another clone after the dirty pages associated with the file are flushed to the persistent store. Figure 4.7 illustrates an example of the sequence of operations during an mmap and msync.

The prototype is implemented as a standalone library that runs on

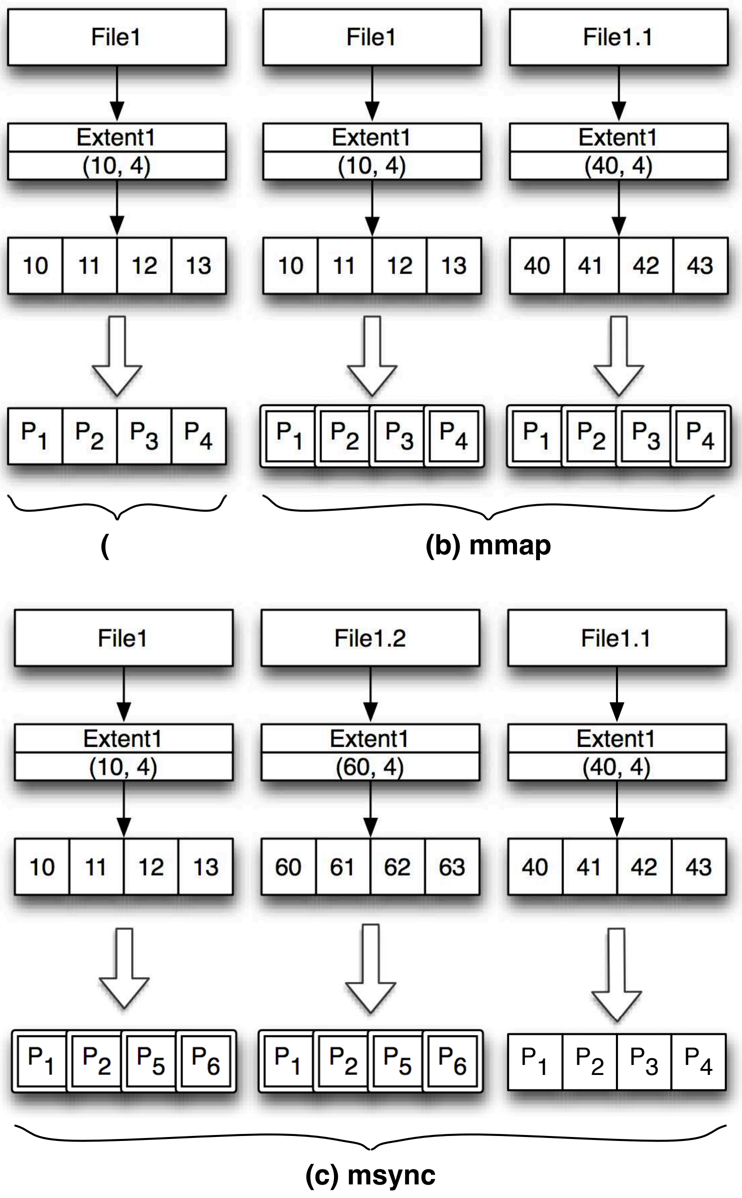

Figure 4.7: **Checkpointed mmap in DirectFS.** *The figure above illustrates the use of checkpointed mmap. The moment we mmap File1, the first clone of the file File1.1 is created by invoking the range clone method. The cloned file File1.1 is identical to the file before it was mmap-ed (Fig b). After the file is modified, we issue an msync that results in the creation of a cloned file File1.2 (Fig c).*

top of DirectFS. The library is around X lines of code and modifications to DirectFS is around 5 lines of code. We modified msync to clone file ranges.

4.7.2 Atomic Writes through Clones in MySQL

Atomic writes is a special primitive exposed by the standard Fusion-io FTL. With the help of atomic writes, the application can write to a set of disjoint addresses and still ensure an all or nothing semantic i.e., either all the writes were successful or none of them were. Fusion-io FTL achieves atomic writes with the help of a log structured device underneath [37, 92]. Unfortunately, every other persistent store can not provide the same type of interface as the log can. Clones can help alleviate this condition.

With the help of range moves, we can redirect the vector of writes to a separate scratch location and once all the writes are successfully completed, we can issue an atomic vectored range-move operation to clone all the ranges from the scratch space to their respective locations. Figure 4.8 shows an example of such an atomic write through range moves.

MySQL uses a double write buffer to ensure atomicity of transactions. Upon a transaction commit, MySQL writes all the dirty pages associated with the transaction to a double write buffer. A double write buffer is a circular buffer used to take in writes from committed transactions. Once all the writes to the double write buffer are complete, MySQL starts to re-issue the writes to the actual physical locations i.e., the double write buffer acts as a journal to ensure consistency and atomicity of transactions. The presence of the double write buffer reduces the throughput available by requiring two writes instead of one. In the case of a flash based backing medium, it also reduces the device lifetime by half since it is doubling the writes to the device. With the

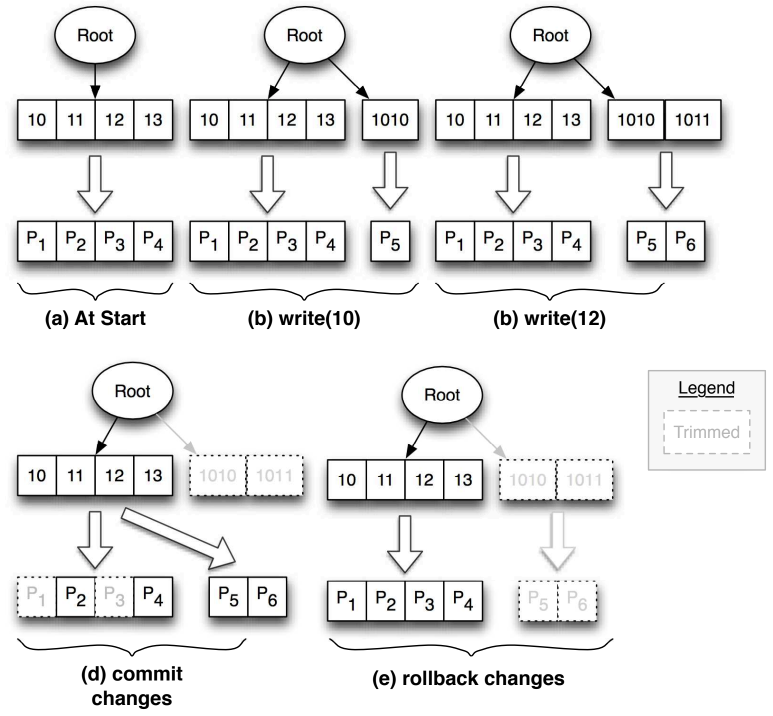

Figure 4.8: **Atomic Writes through Range Moves.** *The figure illustrates the use of range clones and moves to accomplish atomic writes. Fig a shows the state of four blocks(logical addresses 10-13) at the start of an atomic write. Block 10 is overwritten, but we redirect that write to a new location 1010 as shows in Fig b. Similarly, block 12 is overwritten to new location 1011 (Fig c). In the event of a commit (Fig d), we can issue a range move from location 1010 and 1011 to locations 10 and 12 respectively and persist the new data. In case of a rollback, all we need to do is discard data found at locations 1010 and 1011 (Fig e)*

help of atomic writes through clones, we can get rid of the double write buffer in MySQL and still work on top of non-log structured medium.

The implementation effort involved around 200 lines of new code and around 10 lines of modifications in the MySQL 5.5.29 source code.

4.7.3 Snapshots through Clones

Creation of a snapshot is relatively simple and similar to the creation of a file-level snapshot in DFS. We identify the range of blocks that represent a volume or partition and clone the range into a new virtual range (this feature has not yet been fully implemented). We provide a description of the technique to help compare it with the approach adopted by ioSnap.

Creating a snapshot of a volume through clones is significantly different from ioSnap described in Chapter 3. The two major differences are during accessing snapshots and during writes to snapshotted data. Explicit activation is not required since the logical to virtual mappings are always available in the top level translation layer. But, this comes at a price: creating a snapshot in ioSnap was trivial (increment the epoch and write a note). With clones, we have to copy the full range of mappings from the source volume to the destination volume and this operation has to occur atomically. Such a large copy of mappings can be very slow and impede on-going operations. We can optimize this by bringing in a lazy copy scheme and we have discussed some of the details in Section 6.2.2.

The second difference arises during regular write operations. When two logical ranges point to the same virtual range (after a clone triggered by a snapshot), a write operation on one triggers splitting of the clone. Splitting a clone involves allocating a new virtual address and updating the logical to virtual mapping. Though activation are instantaneous, write performance is affected by the presence of clones. On the other hand, ioSnap had trouble with activation stemming from the fact that we never persisted snapshot metadata. Finally, the clones approach also does not interfere with the underlying segment cleaner unlike ioSnap.

Thus, snapshotting through clones offers a different design point when compared to ioSnap with its own advantages and disadvantages. The choice of the system to employ would depend on the workload and how often one expects to use the backup.

4.8 CONCLUSIONS

Through clones, we have put forth the need for newer interfaces when accessing flash devices. Flash is not just another block device. It is different and potentially more powerful. Of course, flash devices can deliver higher throughput, lower latencies with the traditional block interface. But the real power of flash lies with the two key concepts that make flash storage feasible: the flash translation layer (FTL) and the log structured storage device.

Any new interface for flash must be able to leverage the characteristic of flash: the FTL, which translates logical addresses to physical addresses, virtualizes address space, and the log, by definition, virtualizes time. The new interface proposed (in the form of the clones subsystem) exposes flash' ability to virtualize as a fundamental and native operation. We presented three low level interfaces namely, range clone, move and merge. We also demonstrated the usefulness of these interfaces in a large variety of usecases ranging from file snapshotting, file copies, deduplication and atomic writes. Table 4.4 presents a summary of the use-cases.

The new interfaces are not restricted to Fusion-io's devices, but are applicable to any FTL implementing a log-structured storage system. These interfaces represent a small step towards an eventual restructure

	Use Case	API Leveraged	Type of Operation
DirectFS	File Snapshots and Copy	Range Clone	Clone source file to destination file
	Address Space fragmentation	Range Move	Move fragments to contiguous range
	File Dedup	Range Clone	Clone files and trim
MySQL	Atomic Writes	Range Move	Write trasaction to and move to dest.
Volume Backup(*)	Snapshots	Range Clone	Clone the volume to a dest. volume

Table 4.4: **Summary of Clones use-cases.** *The table above presents a summmary of the various use cases for the new interfaces. The table describes the application, the specific use case, the API used and brief description of the operation involved. (*) Snapshots needs further development for better performance.*

of interactions with non-volatile memory. More importanty, they allow applications to step beyond the limitations of the block interface and express their intentions in a flash-aware manner.

5 RELATED WORK

In this thesis, we have covered two major topics in data management over NAND flash. Snapshots are necessary for enterprises who want to retain data for both business and legal reasons, and flash awareness is vital. Secondly, with falling costs, flash is becoming more prevalent in enterprises and application writers are bottlenecked by the lack of flexibility in the standard read/write interface. Flash enables newer ways of interacting with data and we have explored interfaces that would allow these interactions.

We now look at other approaches people have adopted to solve these problems. The rest of the chapter is organized as follow. First, we describe other approaches to creating snapshots in storage systems(Section 5.1). Next, we explore other systems that virtualize address space and time(Section 5.2).

5.1 SNAPSHOTS

Snapshots are point-in-time representations of the state of a storage device. Typical storage systems employ snapshots to enable efficient backups and more recently to create audit trails for regulatory compliance [96]. Many snapshot systems have been designed in the past with varied design requirements: some systems implement snapshots in the file system while others implement at the block device layer. Some systems have focused on the efficiency and security aspects of snapshots, while others have focused on data retention and snapshot access capabilities. In this section we briefly overview existing snapshot systems and their design choices. Table 5.1 is a more extensive (though not exhaustive) list of snapshotting systems and their major design choices.

System	Type	Metadata Efficiency	Consi– tency
WAFL[51]	COW FS	-	Yes
btrfs[6]	COW FS	-	Yes
ext3cow[96]	COW FS	-	Yes
PersiFS[99]	COW FS	-	Yes
Elephant[109]	COW FS	Journaled	Yes
Fossil[103]	COW FS	-	Yes
Spiralog[132]	Log FS	Log	Yes
NILFS[67]	Log FS	Log	Yes
VDisk[136]	BC on VM	-	No
Peabody[87]	BC on VD	-	Yes
Virtual Disks [98]	BC on VD	-	Yes
BVSSD[53]	BC	-	No
LTFTL[118]	BC	-	No
SSS[116]	Object COW	Journaled[113]	Yes
ZeroSnap[102]	Flash Array	-	Yes
TRAP Array[140]	RAID COW	XOR	No

Table 5.1: **Versioning Storage Systems.** *The table presents a summary of several versioning systems comparing some of the relevant characteristics including the type (copy-on-write, log structured, file system based or block layer), metadata versioning efficiency and snapshot consistency. (BC:Block COW, VD:Virtual Disk TSD:Typesafe disk)*

5.1.1 Block Level or File System?

From the implementation perspective, snapshots may be implemented at the file system or at the block layer. Both systems have their advantages. Block layer implementations let the snapshot capability stay independent of the file system above, thus making deployment simpler, generic, and hassle free [136]. The major drawbacks of block layer snapshotting include no guarantees on the consistency of the snapshot, lack of metadata storage efficiency, and the need for other tools outside the file system to access the snapshots [87]. ioSnap is also a block level snapshotting solution and it can also potentially suffer from application data consistency issues. The storage efficiency problem is implicitly solved: the log is always append-only and accessing the snapshot is made simple by exposing it as a block device.

File system support for snapshots can overcome most of the issues with block level systems including consistent snapshots and efficient metadata storage. Systems like PureStorage [102] , Self Securing Storage [116] and NetApp filers [51] are stand-alone storage boxes and may implement a proprietary file system to provide snapshots. File system level snapshotting can give rise to questions on the correctness, maintenance, and fault tolerance of these systems. For example, when namespace tunnels [51] are required for accessing older data, an unstable active file system could make snapshots inaccessible. Some block level systems also face issues with system upgrades (for example, Peabody [87]) but others have used user level tools to interpret snapshotted data, thus avoiding the dependency on the file system (for example, VDisk [136]).

5.1.2 Metadata: Efficiency and Consistency

Any snapshotting system has to not only keep versions of data, it must also version the metadata, without which accessing and making sense of versioned data would be impossible. Metadata can be easily interpreted and thus changes can be easily compressed and stored efficiently. For example, while versioning a file, the inode also changes: modification or access times are updated, size adjusted, etc. Versioning the inode does not require a copy of the old inode, but only requires the name and value of the field that changed [113]. With the knowledge of the change, the inode modifications do not necessarily require a copy of the old inode, but only require the value of the field that changed [113]. Other systems like BVSSD [53] and LTFTL [118] focus on smaller SSDs (up to 100s of GB), and hence are not really concerned with metadata efficiency and end up persisting FTLs for snapshots. Log structured file systems that support snapshotting like NILFS [67] also end up maintaining a lot of metadata to allow snapshot access. ioSnap minimizes its metadata overheads by only keeping a copy of the modified validity bitmaps at the granularity or 4KB memory pages. The only time when ioSnap suffers excessive memory overheads are during activation: the secondary tree constructed does not share any nodes with the primary (or active) tree, even though portion of the tree may represent the same data on the log.

Maintaining metadata consistency across snapshots is important to keep older data accessible at all times. The layer where snapshotting is performed, namely file system or block layer, determines how consistency is handled. File system snapshots can implicitly maintain consistency by creating new snapshots upon completion of all related operations, while block layer snapshots cannot understand relationships between blocks and thus consistency is not guaranteed. For example, while snapshotting, a file, the inode, the bitmaps, and the indirect blocks may be required for a consistent image. With writes to these blocks not always in order, a block level solution may have snapshotted some of the structures before a crash. Without any idea about the relationships between these structures, a clean recovery is impossible. Systems like Peabody [87] depend on the file system's crash recovery mechanisms to ensure consistency. ioSnap relies on the system triggering the snapshot to help ensure consistency. ioSnap does not try to provide any guarantees of consistency of application data, it is the job of the application to determine when data is consistent for a snapshot to be taken. Some file systems may provide utilities that assists snapshot creation like the xfs_freeze [139] by blocking all new operations to the file system.

5.1.3 Snapshot Access and Cleanup

In addition to creating and maintaining snapshots, the snapshots must be made available to users or administrators for recovery or backup. Moreover, users may want to delete older snapshots, perhaps to save space. Interfaces include namespace tunnels (for example, an invisible subdirectory to store snapshots, appending tags or timestamps to file names as used in WAFL [51], Fossil [103] or Ext3cow [96]), user level programs to access snapshots(for example, VDisk [136]). ioSnap also indirectly provides a namespace tunnel by exposing a new block device when a snapshot is activated. The file system on this block device can be mounted and used like any other block device. We believe this approach is cleaner and safer: keeping the snapshot and active data on two different block devices can prevent accidental overwrites or inconsistencies.

Older snapshots (including ones outside the detection window and ones that have been backed up) may be deleted to recover space. The deletion of older snapshots introduces a substantial security vulnerability, where a malicious user may intentionally delete snapshots to cover her tracks [116]. Thanks to the build-in segment cleaner, ioSnap's deletion routines are very simple. The security aspects of snapshot deletion are outside the scope of this work, though we feel that some application involvement is necessary to solve this problem.

5.2 CLONES

Clones are a new way of looking at things that can be done at the block layer. Other research papers and commercial software have also looked at techniques to virtualize both time and space. In this section, we present a summary of the approaches others have adopted to tackle similar problems.

5.2.1 Virtualizing Space and Time

Systems have approached virtualizing the user visible address space and logical time in a multitude of ways. Snapshots represent a natural way to virtualize the address space over time. Since the previous section dealt with snapshots, we will focus on other approaches.

5.2.1.1 Operating Systems

Operating systems (in particular file systems) have been leveraging the ability to share resources [22] through virtualization [57, 112, 129]. In this section, we briefly go over some of the techniques observed in operating systems that allow resource virtualization.

Virtualization software (e.g., VMWare ESX [131], VirtualBox [91], Disco [24]) have been built for the primary reason of resource sharing. These systems eliminate duplicates (through indirection and copy-on-write) to reduce memory and employ copy-on-write snapshots to checkpoint system state [8].

Simple file system techniques like soft-links and hard-links have been around for a very long time and help virtualize the file contents by giving them two different names [7]. The links represent the same piece of data and modifying one file would automatically modify the other.

More advanced address space sharing techniques are employed in copy-on-write [36, 97] file systems like ZFS [21] and Btrfs [6, 135]. The copy-on-write nature of these file systems allow easy sharing of blocks between various entities (files, volumes etc). ZFS clones [5] are volume level copy-on-write, writable snapshots. Semantics of ZFS clone creation requires creation of a volume snapshot, which is then cloned to produce a new writable volume with the same contents as the snapshot. Btrfs provides ioctls to clone a range of addresses. Btrfs, like other copy-on-write file systems, employs reference counting [105] to keep track of the number of pointers to each block. When a write is issued to a block that is shared between clones, a copy is created, reference counts are adjusted and writes are redirected to the copy. These file system level clones come closest to the clones we build.

File systems like MapFS [137] have considered exposing the address space mappings in a more explicit manner. MapFS provides three new interfaces that exposes the file to blocks mapping, allows remapping blocks to other files and deleting mappings from a file. A combination of these interfaces can help optimize many operations include *cp*, deduplication [59, 143], in-place deletion of file ranges etc. MapFS provides a subset of functionality the clones system provides and is file system specific by design. The clones system is file system independent and can provide the same functionality across file systems (provided they are all mounted on a partition of the clone device).

5.2.1.1.1 Address space fragmentation in COW systems Any system that relies on indirection to provide functionality and performance must eventually deal with fragmentation of data on the physical media. A copy-on-write file system like Btrfs [6], WAFL [51] and ZFS [21] rely heavily on the indirection introduced by the copy-on-write mappings between file system namespace and the physical blocks to implement consistency, snapshots and clones. A side effect of copy-on-write is the fragmentation of the address space when blocks are copied to new locations. To ensure the same level of performance (sequentiality of reads) and reduce overheads(contiguous data requires smaller metadata), these file systems require defragmentation [106]. For the same reasons, the indirection layer we built within the clone system also requires defragmentation and some thoughts on it are discussed in Chapter 6.

5.2.1.2 Transactional Systems

Databases have traditionally provided ACID guarantees [104] with the help of techniques like locking [35, 44](*I*solation), write ahead logging [84](*A*tomicity and *D*urability) and rollback [84](*C*onsistency). Modern systems have adopted shadow paging [18, 74, 141] as a replacement for WAL to avoid complex recovery protocols. Multi-version concurrency control is one such approach that is gaining traction in the database community [16, 25, 65, 69], both traditional and main-memory. Traditional single version locking only works for short transactions with no hot-spots. Multi-version concurrency control requires competing threads of execution to work on their own copy of data (usually versioned with the help of a sequence number [1]. The commit of a transaction requires determining the final state of the data based on the various transactions that have also been working on the shared data. The details of schemes can be found at [69]. The transaction begin and commit operations, which copy and finally merge versions, are equivalent to the clone and merge primitives we introduced. In the future, with the help of such primitives in memory and on persistent storage, we should be able to simplify concurrency control in databases.

Systems like Stasis [111] have created a transactional storage engine that attempts to provide a generic library to allow applications to leverage primitives like atomic multi-page writes, transactions etc. It borrows ideas like write-ahead logging and zero-copy [114] to provide high performance primitives that upstream applications like databases can leverage and achieve significant performance boosts(3x improvement in object persistence while cutting memory requirements by half). The clones interface is a superset of the Stasis interface, allowing the same set of interfaces to be built on top of the block device directly instead of pushing all the intelligence into a library.

5.2.2 Use cases

The use cases we described in Section 4.7 have been implemented in the past with different approaches and with varying levels of success. We briefly go over some of the approaches that stand out and compare the clones approach to those.

5.2.2.1 Zero-Copy "cp"

Btrfs [4] implement a zero-copy "cp" by leveraging the clone ioctl. The clone ioctl allows copying of the logical (name space) to physical (block) mappings to create a new mapping. With the help of the clone ioctl, copying a file become a trivial operation requiring no physical copy of blocks. Once the logical to physical mappings are copied to the new file, the underlying blocks are implicitly shared between both files. The copy-on-write mechanism kicks in during writes to split the clones (assign individual blocks to the files on demand). The clones file copy and the file-level snapshot use cases are identical to the clone ioctl of Btrfs.

5.2.2.2 MapFS: Exposing address space mappings

MapFS [137] is a strict superset of the capabilities of the native Btrfs. MapFS leverages the Btrfs clone ioctl [4] to implement most of its interfaces (read a file's address space mapping, remap file blocks and delete mappings). The file copy and file level snapshot features can be implemented easily using the MapFS interfaces. We believe our interfaces are more generic and not tied to one specific file system making them more widely applicable.

5.2.2.3 Atomic mmap

Failure-Atomic msync (FAMsync) [93] deals with the same problem as our implementation of the checkpointed msync use case discussed in Section 4.7.1.5. FAMsync is implemented by passing a new flag to indicate the requirement of atomicity during mmap. Pages belonging to such mmap-ed files are now marked with a new page flag that prevents asynchronous write-back of dirty pages. Finally, to ensure atomicity of write-backs during an msync call, FAMsync leverages the Linux jbd layer [28, 64] in data journaling mode. Data is first written to the journal and only on successful completion is it written in-place. The clone approach is non-invasive, requiring no changes to the operating system.

5.2.2.4 Deduplication

Deduplication (online and offline) has been a hot topic of research and commercial importance [11, 34] over the last decade [59, 143]. Research has spread topics ranging from data fingerprinting [63, 72], block size optimizations [31], backup efficiency [143], optimizations for SSDs [80] to deduplication of virtual machine images [60]. The clones work is meant to illustrate the usefulness of clones for online deduplication and not as a replacement for other deduplication algorithms or systems.

6 FUTURE WORK

The work described in Chapter 3 and 4 presents the key ideas behind creating snapshots on flash and exposing new interfaces to access and manipulate data on flash. In this section, we explore some of the future directions we plan to pursue. We present high level descriptions of some of the clones infrastructure pieces that are needed for completeness. We also discuss some of the techniques we believe will help produce better snapshots and introduce the notion of snapshot ting through clones.

6.1 CLONES INFRASTRUCTURE

The clones infrastructure requires two important pieces to guarantee completeness: the garbage collector and crash recovery.

6.1.1 Garbage Collection

Garbage collection in the presence of two levels of indirection is more complex than before (Section 2.2.3). We need to garbage collect physical blocks that are not used anymore as well as virtual addresses that are no longer pointed to by logical addresses. Fortunately, with the separation between the two levels, the garbage collector for the physical media remains unmodified. The job of the upper layer is to clearly identify virtual addresses that are not used anymore and issue trims to the lower level. The trims will allow the lower level to correctly garbage collect physical blocks.

6.1.2 Crash Recovery

A clean shutdown of the system would result in the secondary map (or indirection layer) being persisted to the underlying media, which could be read back and the system initialized. On an unclean shutdown, none of the mappings are persisted. Fortunately for us, the lower level of indirection is automatically reconstructed using the procedure described in Section 2.2.4. Our job now is to reconstruct the upper level of indirection.

We have persisted notes that correspond to the logical to virtual map and the range operations performed over time. Reconstructing the primary level of indirection involves reading and sorting the address notes and applying the range operations on those (in logical time order). The overall process would resemble the sort and rebuild approach taken by the VSL driver.

6.2 BETTER SNAPSHOTS

Snapshots for flash based devices are important and ability to create them in a flash-aware manner is paramount. The approach we adopted in Chapter 3 has several advantages (simplicity, performance, light-weight), but also imposes overheads on background tasks (segment cleaning, activation). We present some of the high level thoughts we have on how to handle these drawbacks. Next, we describe an alternate approach to snapshotting a volume (or device) with the help of clones.

6.2.1 Mitigating ioSnap's Inefficiencies

The choices we made in Section 3.2 while designing ioSnap have ensured the driver's performance is not compromised in the presence of snapshots. Snapshot creation is very fast and largely invisible to foreground work. Unfortunately, deferring snapshot operations comes at a price. Clearly, activation is an expensive process in terms of space and time. In the worst case, activations may spend tens of seconds reading the full device and in the process consume memory to accommodate a second FTL tree, which shares no memory with the active tree. Rate-limiting can help control impact on foreground processes (both during activation and segment cleaning) making these background tasks either fast or invisible, but not both.

ioSnap's design choices represent just one of many approaches one may adopt while designing snapshots for flash. Clearly, some issues with our design like the activation and COW overheads need to be addressed. Most systems allow instantaneous activation by always maintaining mappings to snapshots in the active metadata [6, 51, 103, 118], but this approach does not scale. Activation overheads may be reduced by precomputing some portions of the FTL and checkpointing on the log. The segment cleaner may also assist in this process by using policies like hot/cold [107] to reduce epoch intermixing, thereby localizing data read during activation.

Finally, keeping snapshots on flash for prolonged durations isn't necessarily the best use of the device. Thus, schemes to destage snapshots to archival disks are required. Checkpointed (precomputed) metadata can hasten this process by allowing the backup manager identify blocks belonging to a snapshot.

6.2.2 Snapshots through Clones with Lazy Loading

Creation of a snapshot is relatively simple and similar to the creation of a file-level snapshot in DirectFS. We identify the range of blocks that represent a volume or partition and clone the range into a new virtual range. The only major difference between the file-level clone and a snapshots is that we do not need to activate the snapshot the moment we create it. We only activate when the backup needs to be accessed. A file-level snapshot in DirectFS invokes a range clone operation, which implicitly copies the source mappings to the destination

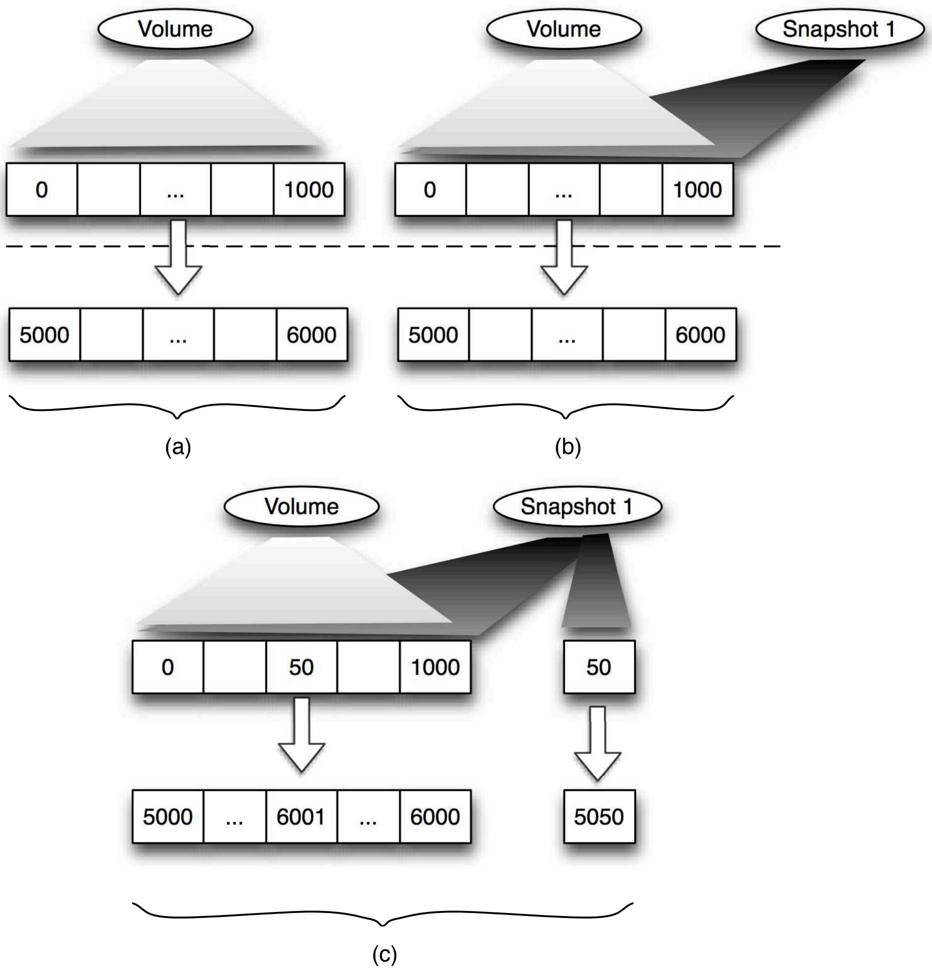

Figure 6.1: **Lazy Activation of Snapshotted Data.** *The figure above illustrates how snapshots of large volumes can benefit from lazy activation. The typical range clone interface results in the source mappings getting copied into the destination address space. But when creating snapshots of volumes, such copies tend to be slow. Instead, we lazily copy mappings only when the source is going to be modified. Fig a represents the volume and Fig b shows the creation of a snapshot that shares the address space with the volume. When block 50 is written, the to retain the original mappings, the snapshot now stores the mapping between address 50 and the old block (5050) and the data is written to the new location 6001.*

mappings. Such a copy, spanning the entire device, would make the overall process of snapshot creation very slow. Moreover, the mappings are only required during snapshot activation or when the clone needs to be broken: either source or destination range is modified.

So, the range clone primitive will provide a delayed or on-demand activate option. When on-demand activate is specified, the clone layer lazily loads mappings whenever needed. Figure 6.1 illustrates an example of the use of lazy activation for snapshotted data.

Deletion of a snapshot is simple: we only need to trim the virtual ranges corresponding to the snapshotted data. The garbage collector takes care of releasing all the unused address ranges and the unused data from the drive underneath.

6.3 BETTER WRITES

One of the biggest challenges faced by file systems and applications is small random writes. Disks have been suffering this problem for a very long time due to the seeks involved during the random writes. Log structures file system [107] introduced the notion of logging to alleviate the performance penalty of small random writes. RAIDs also suffers from the small write problem and addition of by spreading the log over a RAID [49]. Flash based storage devices also build a log and thus can address this problem in theory. But, in reality, random write performance in flash is orders of magnitude lesser than the sequential counterpart [43]. The reason for this behavior is the fact that small random writes do not parallelize well over the multiple NAND banks present in typical flash array [40]. Researchers have tried to address this problem by over-provisioning physical space and managing data placement to minimize the impact small random writes [27].

Clones can again help mitigate part of this problem. The incoming sequence of random writes can be remapped to a sequential address stream borrowed from the virtual space. The underlying flash layer thus only sees a sequential stream of data and it can write this out in a highly performant manner. At periodic intervals, we can issue a range move from the virtual range to the original range, thus keeping the data available in the user specified address range. To ensure correctness, we also have to persist the mappings between the incoming writes and the virtual address to which it was written.

6.4 SUMMARY

The advent of flash has uncovered several opportunities that require us to rethink how we build applications on top of flash. In addition to the ideas demonstrated in Chapter **??** and 4, we have presented infrastructure pieces necessary for completeness and more ways to leverage the new interfaces. Better interfaces lead to better applications and we believe the clones system is just the tip of the iceberg.

7 CONCLUSIONS

Modern storage systems have been well served by the block interface. The simplicity of the interface allowed wide-spread adoption and provided an abstraction that allowed software layers above it to thrive. Unfortunately, all good thing come to an end. Flash is a different medium and trying to make it behave only like a block device is unproductive.

Several years of work has gone into developing software and abstractions on top of block devices. When run on top of flash, some abstractions are elegant and efficient (for example, name-space through file systems, buffering through the page cache), while others are simply unwieldy and ineffective (for example, data journaling, physical copy of files). Thus, it is important that we revisit some of the old software design choices as well as explore newer ways of interacting with flash, which will enable applications (old and new).

7.1 TRADITIONAL FEATURES: SNAPSHOTS THROUGH IOSNAP

Traditionally, snapshots are performed by the file system or the volume manager. Through ioSnap, we have incorporated the ability to snapshot data on flash, in a performant and non-invasive manner, leveraging the properties of the FTL. ioSnap takes a different approach and focusses its design on keeping common operations fast, even in the presence of snapshots and the segment cleaner. This comes at the expense of accessing snapshots.

With the help of simple rate-limiting, we able to control the extent to which background tasks affected foreground operations. To perform meaningful rate-limiting, we needed information that was only available within the FTL, thus supporting our decision to implement snapshots in the FTL.

7.2 RETHINKING INTERFACES: CLONES

Though flash exposed as a block device, can deliver significant performance gains, it is stifled by the block interface. The new interfaces understand flash and leverage things that flash can uniquely offer. We focus on the ability of flash to virtualize address spaces (through the FTL) and virtualize time (through the log). We demonstrate the usefulness of these interfaces in the realm of file-systems and applications (MySQL). These interfaces can be leveraged to simplify application's tasks and in turn help improve the lifetime of the device. For example, a file copy or a snapshot translates to a simple clone of the address space instead of the traditional block by block copy. Also, MySQL can achieve its consistency goals by leveraging atomic writes instead of issuing two writes.

The new interfaces represent an important step towards tailoring applications to work well on top of flash. Applications need to understand the difference between running on flash and hard-disk. The device characteristics are different and trying to duplicate (or interfere with) work done by the device can prove detrimental to performance and lifetime. Thus, these interfaces demonstrate the need for a rethink of the protocols for persisting data on top of solid-state devices.

7.3 LESSONS LEARNED

Through the course of understanding the systems we built, we have, time and again, realized that our designs are driven by a small set of principles that have shaped our thought process. We list some of these lessons and why we feel they are important.

- **The log is your friend**

  The log is a powerful construct. File systems and databases have used logging to ensure consistency for a very long time and they have demonstrated the usefulness of the log. Flash devices are forced to implement a log and this represents a very powerful opportunity. Leveraging the log can simplify design, but ignoring is could lead to poor performance.

- **Be aware of the medium**

  The differences between flash and hard-disk and its consequences are well known and well documented. Unfortunately, adding new features to the FTL also has consequences. For example, the garbage collector must always be treated carefully: data can be reordered anytime. The FTL also requires metadata, which must be recreated after a crash (in addition to the metadata of filesystems that may be running on top). Any new feature must always take into consideration the impact on these two components.

- **Leverage the indirection**

  Indirection has always played a significant role in various portions of the operating systems. Now, flash is also forced to turn to indirection, in the form of the FTL, to provide a practical storage medium. Typically, indirection layers are always associated with overheads. But as it turns out, the presence of the FTL is a powerful abstraction that applications can leverage. We need to explore more ways of exposing and manipulating the FTL mappings.

- **Think outside the "block"**

  For several years, the block interface has served the storage industry well. The simple interface allowed features to be added in software and given the slow nature of the backing media, inefficient software implementations were never the bottleneck. But flash is a different beast and we should never restrict our thought process to the block device. Newer interfaces are needed for devices with an intermediate translation layers.

- **Know your workloads**

  Though knowledge of the workload is a generic principle behind almost all systems, it is more important with respect to flash. Flash is primarily used for its high performance (without which the total cost of operations would not compare). Addition of new features that affect performance is frowned upon, unless the benefits can really outweigh the loss. Thus, knowledge of the workloads must drive design of new features with minimal impact on performance.

Flash is here to stay or at least until the next big thing replaces it. But, with the next major paradigm shift at least a decade away, sticking to old interfaces designed for hard-disks is no longer acceptable. In this thesis, we have demonstrated the need for a rethink of the layer at which storage system features are implemented and also explored a possible enhancement to the flash interfaces. Flash is becoming an important part of storage systems and hence, native interfaces and redesigned software represent the way forward.

REFERENCES

[1] MVCC: Transaction IDs, Log Sequence numbers and Snapshots. http://www.mysqlperformanceblog.com/2007/12/19/mvcc-transaction-ids-log-sequence-numbers-and-snapshots/.

[2] http://download.micron.com/pdf/datasheets/flash/nand/1gb_nand_m48a.pdf.

[3] http://maltiel-consulting.com/NAND_vs_NOR_Flash_ Memory_Technology_Overview_Read_Write_Erase_speed_for_SLC_MLC_semiconductor_consulting_expert.pdf.

[4] Save disk space on Linux by cloning files on Btrfs and OCFS2. https://blogs.oracle.com/OTNGarage/entry/save_disk_space_on_linux, 2010.

[5] Overview of ZFS Clones. http://docs.oracle.com/cd/E19253-01/819-5461/gbcxz/index.html, 2010.

[6] Btrfs Design. http://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-design.html, 2011.

[7] GNU Core Utils. http://www.gnu.org/software/coreutils/, 2013.

[8] Understanding virtual machine snapshots in VMware ESXi and ESX. http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1015180, 2013.

[9] Nitin Agarwal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[10] S. Agarwal, D. Borthakur, and I. Stoica. Snapshots in hadoop distributed file system. *UC Berkeley Technical Report UCB/EECS*, 2011.

[11] Carlos Alvarez. Netapp deduplication for fas and v-series deployment and implementation guide. Technical report, Technical Report TR-3505, NetApp, 2011.

[12] Andrew Ku. Endurance Testing: Write Amplification And Estimated Lifespan. http://www.tomshardware.com/reviews/ssd-520-sandforce-review-benchmark,3124-11.html, 2012.

[13] Apple. OSX Time Machine. `http://www.apple.com/findouthow/mac/#timemachinebasics`, 2007.

[14] Joel Bartlett, Wendy Bartlett, Richard Carr, Dave Garcia, Jim Gray, Robert Horst, Robert Jardine, Doug Jewett, Dan Lenoski, and Dix McGuire. The Tandem Case: Fault Tolerance in Tandem Computer Systems. In Daniel P. Siewiorek, and Robert S. Swarz, editors, *Reliable Computer Systems - Design and Evaluation*, chapter 8, pages 586–648. AK Peters, Ltd., October 1998.

[15] Ben Treynor. Gmail Outage. http://gmailblog.blogspot.com/2011/02/ gmail-back-soon-for-everyone.html, 2011.

[16] Philip A Bernstein, and Nathan Goodman. Multiversion concurrency controlâŁ"theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[17] Eric J. Bina, and Perry A. Emrath. A Faster fsck for BSD Unix. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '89)*, San Diego, California, January 1989.

[18] Dina Bitton, and Jim Gray. Disk shadowing. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB 14)*, pages 331–338, Los Angeles, California, August 1988.

[19] Matias Bjorling, Philippe Bonnet, Luc Bouganim, Niv Dayan, et al. The necessary death of the block device interface. In *6th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[20] Jeff Bonwick. RAID-Z. http://blogs.sun.com/bonwick/entry/raid_z, 2005.

[21] Jeff Bonwick, and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[22] Daniel P. Bovet, and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2006.

[23] Aaron B. Brown, and David A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 263–276, San Diego, California, June 2000.

[24] Edouard Bugnion, Scott Devine, and Mendel 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.

[25] Michael J Carey, and Waleed A Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems (TOCS)*, 4(4):338–378, 1986.

[26] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(4):837–863, 2004.

[27] Brice Chardin, Olivier Pasteur, and Jean-Marc Petit. An ftl-agnostic layer to improve random write on flash memory. *Database Systems for Adanced Applications*, pages 214–225, 2011.

[28] Ying CHEN, and Hong-sheng XI. Performance optimization of journaling file system based on jbd. *Computer Engineering*, 8: 020, 2010.

[29] Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings Joint NASA and IEEE Mass Storage Conference*, volume 3, 1998.

[30] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[31] Cornel Constantinescu, Jan Pieper, and Tiancheng Li. Block size optimization in deduplication systems. In *Data Compression Conference, 2009. DCC'09.*, pages 442–442. IEEE, 2009.

[32] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 87–100, San Francisco, California, December 2005.

[33] Ming Du, Yan Zhao, and Jiajin Le. Using flash memory as storage for read-intensive database. In *Proceedings of the 2009 First International Workshop on Database Technology and Applications*, DBTA '09, 2009.

[34] EMC Datadomain. EMC Datadomain. http://www.emc.com/domains/datadomain/index.htm, 2013.

[35] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[36] Francisco Javier Thayer Fábrega, Francisco Javier, and Joshua D Guttman. Copy on write. 1995.

[37] Fusion-io. Atomic Writes Accelerate MySQL Performance. http://www.fusionio.com/blog/atomic-writes-accelerate-mysql-performance, 2011.

[38] Fusion-io. DirectFS. http://www.fusionio.com/webinar/iomemory-sdk-directfs-native-filesystem-services, 2012.

[39] Fusion-io. FusionIO Direct Cache. `http://www.fusionio.com/products/directcache`, 2012.

[40] Fusion-io. FusionIO IoDrive2. `http://www.fusionio.com/products/iodrive2-duo/`, 2012.

[41] Fusion-io. FusionIO IoN Data Accelerator. `http://www.fusionio.com/products/ion-data-accelerator/`, 2012.

[42] Fusion-io. FusionIO IOMemory SDK. `http://www.fusionio.com/products/iomemorysdk`, 2012.

[43] Jim Gray, and Bob Fitzgerald. Flash disk opportunity for server applications. *Queue*, 6(4):18–23, 2008.

[44] Jim Gray, and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[45] 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.

[46] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of nand flash memory. FAST'12, 2012.

[47] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 43th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 229–240, Washington, DC, March 2009.

[48] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[49] J.H. Hartman, and J.K. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 29–43, Asheville, North Carolina, December 1993.

[50] Val Henson. The Many Faces of fsck. http://lwn.net/Articles/248180/, September 2007.

[51] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[52] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 10. ACM, 2009.

[53] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. Bvssd: build built-in versioning flash-based solid state drives. SYSTOR '12, 2012.

[54] Gordon F. Hughes, and Joseph F. Murray. Reliability and Security of RAID Storage Systems and D2D Archives Using SATA Disk Drives. 1(1):95–107, February 2005.

[55] Intel. Intel SSD 910 Series. http://ark.intel.com/products/67009/ Intel-SSD-910-Series-800GB-12-Height-PCIe-2_0-25nm-MLC, 2013.

[56] Intel. Intel Solid State Drives. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-ssd.html, 2013.

[57] Bruce L Jacob, and Trevor N Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. *ACM SIGPLAN Notices*, 33(11):295–306, 1998.

[58] James Hamilton. When SSDs Make Sense in Server Applications. http://perspectives.mvdirona.com/2008/10/15/ WhenSSDsMakeSenseInServerApplications.aspx, 2008.

[59] Keren Jin, and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, New York, NY, USA, 2009.

[60] Keren Jin, and Ethan L Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 7. ACM, 2009.

[61] Jonathan Corbet. Block layer discard requests. `http://lwn.net/Articles/293658/`, 2008.

[62] William K. Josephson, Lars A. Bongo, Kai Li, and David 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.

[63] Jürgen Kaiser, Dirk Meister, Andre Brinkmann, and Sascha Effert. Design of an exact data deduplication cluster. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12. IEEE, 2012.

[64] Kedar Sovani. Linux: The Journaling Block Device. http://kerneltrap.org/node/6741, 2006.

[65] Thomas F. Keefe, and Wei-Tek Tsai. Multiversion concurrency control for multilevel secure database systems. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 369–383. IEEE, 1990.

[66] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proceedings of the*

3rd USENIX Conference on File and Storage Technologies, pages 59–62, 2004.

[67] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS OSR*.

[68] Charles M. Kozierok. Overview and History of the SCSI Interface. http://www.pcguide.com/ref/hdd/if/scsi/over-c.html, 2001.

[69] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, 2011.

[70] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, 2008.

[71] M Lenzlinger, and EH Snow. Fowler-nordheim tunneling into thermally grown sio 2. *Journal of Applied Physics*, 40(1):278–283, 1969.

[72] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and storage technologies*, pages 111–123, 2009.

[73] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on*

Operating Systems Principles, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6.

[74] R. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Databases*, 2(1):91–104, 1977.

[75] Lanyue Lu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution.

[76] M-Systems. Two Flash Technologies Compared: NOR vs NAND. http://focus.ti.com/pdfs/omap/diskonchipvsnor.pdf, 2002.

[77] M. Mano. *Digital logic and computer design*. Prentice-Hall, 1979. ISBN 9780132145107. URL `http://books.google.com/books?id=QWZTAAAAMAAJ`.

[78] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[79] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[80] Dirk Meister, and Andre Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (ssd). In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–6. IEEE, 2010.

[81] ST Microelectronics. Wear leveling in single level cell nand flash memories. *Application note AN-1822 Geneva, Switzerland*, 2007.

[82] Micron. Micron Enterprise PCIe SSD. http://www.micron.com/products/solid-state-storage/enterprise-pcie-ssd, 2013.

[83] Micron Technologies. TLC, MLC and SLC Devicesage Analysis. http://www.micron.com/products/nand-flash/tlc-mlc-and-slc-devices, 2013.

[84] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[85] Vidyabhushan Mohan. *Modeling the Physical Characteristics of NAND Flash Memory*. PhD thesis, University of Virginia, 2010.

[86] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea R Stan. How i learned to stop worrying and love flash endurance. In *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, pages 3–3. USENIX Association, 2010.

[87] Charles B. Morrey III, and Dirk Grunwald. Peabody: The time travelling disk. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, 2003.

[88] Nick Allen. Flash-based SSDs are Driving New Standards and Charging Models. http://wikibon.org/wiki/v/ Flash-based_SSDs_are_Driving _New_Standards_and_Charging_Models, 2012.

[89] OCZ. OCZ Solid State Drives. http://ocz.com/consumer/ssd, 2013.

[90] OCZ. SSD Comparison. http://ocz.com/consumer/ssd-guide/ssd-comparison, 2013.

[91] Oracle. VirtualBox. https://www.virtualbox.org, 2013.

[92] Xiangyong Ouyang, David W. Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *HPCA*, pages 301–311. IEEE Computer Society, 2011.

[93] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync (): A simple and efficient mechanism for preserving the integrity of durable data. 2013.

[94] David Patterson, Garth Gibson, and Randy 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.

[95] Percona. Tuning For Speed: Percona Server and Fusion-io. http://www.percona.com/files/presentations/percona-live/london-2011/PLUK2011-tuning-for-speed-percona-server-and-fusion-io.pdf, 2012.

[96] Zachary Peterson, and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, 2005. ISSN 1553-3077.

[97] Zachary Nathaniel Joseph Peterson. *Data placement for copy-on-write using virtual contiguity.* PhD thesis, UNIVERSITY OF CALIFORNIA, 2002.

[98] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: getting beyond the limitations of virtual disks. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06.

[99] Dan R. K. Ports, Austin T. Clements, and Erik D. Demaine. Persifs: a versioned file system with an efficient representation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, 2005.

[100] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[101] PureStorage. PureStorage FlashArray. http://www.purestorage.com, 2013.

[102] PureStorage. ZeroSnapâ„¢ snapshots. http://www.purestorage.com/flash-array/resilience.html, 2013.

[103] S. Quinlan, and J.M.K.R. Cox. Fossil, an archival file server.

[104] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill, 2003.

[105] Ohad Rodeh. Deferred reference counters for copy-on-write b-trees. Technical report, Technical Report rj10464, IBM, 2010.

[106] Ohad Rodeh. Defragmentation mechanisms for copy-on-write file-systems. 2010.

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

[108] Samsung Semiconductors. NAND: Beyond Memory to Storage. http://originus.samsung.com/us/business/oem-solutions/pdfs/2011-03-Mobile-ization-NAND.pdf, 2011.

[109] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding When To Forget In The Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 110–123, Kiawah Island Resort, South Carolina, December 1999.

[110] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 267–280. ACM, 2012.

[111] Russell Sears, and Eric Brewer. Stasis: flexible transactional storage. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 29–44. USENIX Association, 2006.

[112] Abraham Silberschatz, and Peter Galvin. *Operating Systems Concepts*. Addison-Wesley, 1998.

[113] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2003. USENIX Association.

[114] Dragan Stancevic. Zero copy i: user-mode perspective. *Linux Journal*, 2003(105):3, 2003.

[115] Storage Search. Flash Memory vs. HDDs - Which Will Win? http://www.storagesearch.com/semico-art1.html, 2005.

[116] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised system. In *Proceedings of*

the 4th conference on Symposium on Operating System Design & Implementation - Volume 4, OSDI'00, 2000.

[117] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, et al. A 3.3 v 32 mb nand flash memory with incremental step pulse programming scheme. *Solid-State Circuits, IEEE Journal of*, 30(11):1149–1156, 1995.

[118] Kyoungmoon Sun, Seungjae Baek, Jongmoo Choi, Donghee Lee, Sam H. Noh, and Sang Lyul Min. Ltftl: lightweight time-shift flash translation layer for flash memory based embedded storage. EMSOFT '08.

[119] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[120] Sun Microsystems. ZFS Under The Hood. http://www.filibeto.org/~aduritz/truetrue/solaris10/zfs-uth_3_v1.1_losug.pdf, 2006.

[121] Sun Microsystems. MySQL White Papers, 2008.

[122] V. Sundaram, T. Wood, and P. Shenoy. Efficient Data Migration for Load Balancing in Self-managing Storage Systems. In *The 3rd IEEE International Conference on Autonomic Computing (ICAC '06)*, Dublin, Ireland, June 2006.

[123] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[124] The AWS Team. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648/, April 2011.

[125] Tom's Hardware. Intel SSD 335 240 GB Review: Driving Down Prices With 20 nm NAND. http://www.tomshardware.com/reviews/ssd-335-240-gb-benchmark,3332.html, 2012.

[126] Toshiba. NAND vs NOR Flash Memory. http://www.toshiba.com/taec/Catalog/components/ Description/pop_mem_compare.html, 2011.

[127] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[128] Stephen C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/ release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[129] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[130] Violin Memory. Violin Memory PCIe Cards. http://www.violin-memory.com/products/velocity-pcie-cards/, 2013.

[131] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[132] Christopher Whitaker, J. Stuart Bailey, and Rod D. W. Widdowson. Design of the server for the Spiralog file system. *Digital Technical Journal*, 8(2), 1996.

[133] Wikipedia. HIPAA. `http://en.wikipedia.org/wiki/HIPAA`, 1996.

[134] Wikipedia. Sarbanes-Oxley. http://en.wikipedia.org/wiki/ Sarbanes%E2%80%93Oxley_Act, 2002.

[135] Wikipedia. Btrfs. en.wikipedia.org/wiki/Btrfs, 2009.

[136] Jake Wires, and Michael J. Feeley. Secure file system versioning at the block level. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, 2007.

[137] Jake Wires, Mark Spear, and Andrew Warfield. Exposing file system mappings with mapfs. In *Proc. of the 3rd USENIX conference on Hot topics in storage and file systems*, 2011.

[138] David Woodhouse. Jffs: The journalling flash file system. In *Ottawa Linux Symposium*, volume 2001, 2001.

[139] XFS. XFS Freeze. http://linux.die.net/man/8/xfs_freeze.

[140] Qing Yang, Weijun Xiao, and Jin Ren. Trap-array: A disk array architecture providing timely recovery to any point-in-time. *SIGARCH Comput. Archit. News*, May 2006.

[141] Tatu Ylonen. Concurrent shadow paging: A new direction for database research. *Laboratory of Information Processing Science, Helsinki University of Technology, SF-02150, Espoo, Finland*, 1992.

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

[143] Benjamin Zhu, Kai Li, and Hugo 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.