REVAMPING THE SYSTEM INTERFACE TO STORAGE-CLASS MEMORY

by

Haris Volos

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012

Date of final oral examination: 12/10/2012

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

Michael Swift, Assistant Professor, Computer Sciences

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences

Shan Lu, Assistant Professor, Computer Sciences

Jignesh M. Patel, Professor, Computer Sciences

David A. Wood, Professor, Electrical and Computer Engineering

© Copyright by Haris Volos 2012

All Rights Reserved

Dedicated to my parents Michalis and Maro

for their unconditional support.

ACKNOWLEDGMENTS

I am indebted to my father for living, but to my teacher for living well.

— Alexander the Great (356-323BC)

The above quote by Alexander resonates with my feelings about the many teachers in my life but especially echoes my deep gratitude to my wonderful advisor Mike Swift. I thank Mike for his patience, support, and guidance throughout my graduate studies; in particular, for his help in defining my research direction, for gladly sharing his deep technical knowledge and insights, for teaching me how to approach research with patience and clearly present my ideas to others, and for putting up with me whenever I revised a paper half an hour before the submission deadline!

I would then like to extend my sincerest thanks and appreciation to the rest of the Wisconsin faculty I closely interacted with: Mark Hill and David Wood for welcoming me into the Multifacet group, which provided a research-conducive environment, for their invaluable advice on my research and career plans, and for the wonderful time during their group beer outings at the terrace; Shan Lu for collaborating and sharing with me her great insights on concurrency bugs; Remzi Arpaci-Dusseau for always finding the time to discuss my questions and provide feedback; David DeWitt for welcoming me into the Jim Gray Systems Lab during my final year of studies; and, Andrea Arpaci-Dusseau and Jignesh Patel for serving in my doctoral committee and providing useful feedback.

I would like to thank all the Wisconsin system students that I had the pleasure to interact with during the years, including Joy Arulraj, Lakshmi Bairavasundaram, Arini Balakrishnan, Vijay Chidambaram, Thanh Do, Neelam Goyal, Haryadi Gunawi, Tyler Harter, Guoliang Jin, Asim Kadav, Swetha Krishnan, Thanumalayan Pillai, Sankaralingam Panneerselvam, Matt Renzelman, Thawan Kooburat, Mohit Saxena, Sriram Subramanian, Swami Sundararaman, Andres Jaan Tack, Venkatanathan Varadarjan, Wei Zhang, Yiying Zhang, and Yupu Zhang. I would especially like to thank Andres, Neelam, and Sankar for technically supporting me and emotionally suffering with me through the many hard paper deadlines. I would also like

to give special thanks to Asim, Matt, and Mohit for offering me their valuable feedback and advice during lunch and coffee breaks, practice talks, and other gatherings.

I would like to thank the rest of my fellow students I had the pleasure to cross paths with during grad school: past and present Multifacet students, including Arkaprava Basu, Jayaram Bobba, Polina Dudnik, Jayneel Gandhi, Dan Gibson, Joel Hestness, Derek Hower, Mike Marty, Kevin Moore, Lena Olson, Marc Orr, Andy Phelps, Jason Power, Somayeh Sardashti, Rathijit Sen, Muhammad Shoaib bin Atlaf, Nelay Vaish, Cong Wang, Yasuko Watanabe, and Luke Yen, for their daily interactions with me and teaching me about computer architecture; other fellow students, including Siddharth Barman, Theophilus Benson, Jichuan Chang, Joe Chabarek, Hamid Reza Ghasemi, Ahmed Ghanem, Gagan Gupta, Cindy Rubio Gonzalez, Piramanayagam Arumuga Nainar, Srinath Sridharan, and Philip Wells. I would especially like to thank: Jayaram, Kevin, and Luke, whom I had the good fortune to closely collaborate with and learn from during the LogTM days; and Arka, whom not only did I share my office room with over the last four years but also many happy and bitter experiences.

I would also like to thank Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Adam Welc for giving me the opportunity to get early exposure to industrial research as an intern at Intel Labs.

Many other great people I met in Madison also contributed to making my seven year stay more joyful. I would like to thank Aris and Christos for tolerating me as their roommate, Charalambos for sailing with me the waters of lake Mendota, and Mina, Giorgos, Spyros, Ekaterini, Avrilia, Katerina, Cesar, Dimitris, Maria, Joanne, Nicos, Loizos, Panikkos, Kostas, George, Liza, and Vasilis for being close to me and comforting. With some of them I had the great pleasure to travel to idyllic destinations. I thank Aris and Charalambos for our California road trip, Spyros and Charalambos for putting up with me during our Bahamas trip, Aris and Liegh for our camping trip in Yellowstone, and Panikkos for backpacking with me around Chile.

Many thanks to my college friends Michalis, Panos, and Stelios, with whom I embarked on the long journey of grad school, for being supportive and always available for a cheerful

phone or Skype chat. I will always recall joyful memories from our reunions.

Last but not least, to my parents, Michalis and Maro, and my brothers, Yiannis and Stavros, I want to say a big THANK YOU for your unconditional love and support in my endeavors all these years. Even though our numerous trans-Atlantic Skype chats have kept us virtually connected, words cannot explain how much I miss you.

CONTENTS

Contents v

List of Tables vii

List of Figures viii

Abstract ix

1 Introduction 1

1.1 *Motivation* 2

1.2 *Thesis and Contributions* 6

1.3 *Dissertation Organization* 9

2 Background and Related Work 11

2.1 *Storage-Class Memory* 11

2.2 *Transactions* 18

2.3 *Persistence Techniques* 23

2.4 *File System Architecture* 28

3 Mnemosyne: Lightweight Persistent Memory 36

3.1 *Design* 36

3.2 *Implementation* 45

3.3 *Durable Memory Transactions* 54

3.4 *Programming Examples* 57

3.5 *Evaluation* 59

3.6 *Summary* 73

4 Aerie: Application-specific File Systems 74

4.1 *Design* 75

4.2 *Implementation* 84

4.3 File Systems Interfaces on Aerie 97

4.4 Evaluation 101

4.5 Summary 108

5 Conclusions 110

5.1 Summary 110

5.2 Lessons Learned 111

5.3 Future Work 116

5.4 Closing words 118

Bibliography 119

LIST OF TABLES

2.1 Access latency, density, and endurance for various solid-state storage technologies 12
2.2 Approaches to using storage-class memory in system design 14

3.1 Methods for consistently updating persistent memory 41
3.2 Mnemosyne's programming interface 46
3.3 Update throughput for OpenLDAP, Tokyo Cabinet, and Memcached 63
3.4 Performance of Mnemosyne updates and Boost serialization of red-black trees . . 70
3.5 Throughput of base and tornbit RAWLs 70

4.1 Aerie layers and their supported abstractions and operations. 78
4.2 Aerie implementation size 84
4.3 Log records of storage-object operations and their type-specific fields 93
4.4 Storage-object invariants 94
4.5 Latency of common file-system operations 103
4.6 Average and 95-percentile latency to complete one workload iteration 105
4.7 Throughput performance of a multiprogrammed workload 107

LIST OF FIGURES

2.1 Local file-system architectures . . . 30

3.1 Mnemosyne architecture . . . 39
3.2 Example of missing write detection in the RAWL . . . 52
3.3 Example use of a durable memory transaction. . . . 57
3.4 Example use of low-level persistence primitives. . . . 58
3.5 Latency and throughput to flush a cache line . . . 66
3.6 Write latency for a hashtable with durable transactions compared to Berkeley DB 69
3.7 Update throughput for a hashtable with durable transactions compared to Berkeley DB . . . 69
3.8 Decrease in write latency of hashtable with asynchronous over synchronous truncation . . . 71
3.9 Mnemosyne's performance relative to Berkeley DB for different memory access latencies . . . 72

4.1 Aerie architecture . . . 75
4.2 Aerie design components . . . 80
4.3 Log record format . . . 93
4.4 Log fragment corresponding to a file create . . . 93
4.5 Log fragment corresponding to a file write . . . 95
4.6 Performance of concurrent clients reading or writing a 4KB page of the same file 104
4.7 File-system throughput performance . . . 109

ABSTRACT

Emerging storage-class memory (SCM) devices, such as phase-change memory and memristors, provide the interface of memory but the persistence of disks. SCM promises low-latency storage, which can benefit modern applications ranging from desktop applications that frequently flush data to stable storage, to large-scale web applications that perform lots of dependent lookups such as social-network graphs. Existing operating-system interfaces, however, fail to expose the full capabilities of SCM. Under the current storage model, the OS mediates every access to storage for protection and to abstract details of the specific storage device through a driver. This causes unneeded complexity and lower performance for SCM, which can be accessed directly through the memory interface rather than peripherally via I/O requests.

This dissertation revisits the system interface to storage in light of SCM. The central hypothesis is that direct access to SCM from user mode can form the basis for a flexible high-performance storage architecture. Direct access promises much higher performance by avoiding the cost of entering the kernel and removing layers of code. Direct access also enables flexibility by letting applications customize the storage system to their needs so as to avoid paying generic overheads and therefore further improve performance.

The dissertation presents the design, implementation, and evaluation of an operating-system storage architecture based on direct access to SCM. The architecture comprises two parts. The first part, Mnemosyne, exposes *persistent memory* to user-mode programs. Persistent memory enables programs directly store and access common in-memory data structures, such as trees, logs, and hash tables, in regions of SCM (private to each program). While persistent memory enables low-latency flexible storage, it sacrifices the common file-system interface that enables application interoperability by organizing data under a global logical namespace for easy access and protecting data for secure sharing. The second part, Aerie, fills this gap. Aerie exposes a flexible file-system interface to SCM using user-mode libraries that access file data directly through memory. Aerie retains both the benefits of direct access and the sharing and protection features of file systems.

1 INTRODUCTION

Emerging device technologies, such as phase-change memory and memristors, promise high-speed and affordable persistent storage. These devices collectively are termed *storage-class memory* (SCM) as they provide the interface of memory but the persistence of disks [65]. That is, SCM enables low-latency persistent storage than can be accessed directly through the memory interface using ordinary load and store instructions.

Modern applications can greatly benefit from such low-latency storage. In the desktop and smartphone domain, applications that perform frequent flushes to slow storage devices may cause sluggish performance that results in poor user experience [44, 101]. In the data center, web applications, such as search, social networks, and electronic commerce, make complex unpredictable explorations of data over large datasets in response to multiple different-user queries. User queries must be served in real time so these applications often resort to keeping a large fraction of their data in memory for low-latency access. For example, Facebook keeps approximately 75% of its total data size in memory [141] and Bing allocates much of its index in memory [108].

Unfortunately, existing operating systems' storage architecture and interfaces fail to expose the full potential of these device technologies to programmers and applications. Existing operating systems are designed for a strict bifurcation of devices into *memory*, volatile, fast, byte-addressable devices, and *storage*, persistent, slow, block-based devices. Under this storage model, the OS mediates every access to storage for protection and to abstract details of the specific storage device through a driver. This causes unneeded complexity and lower performance for SCM, which can be accessed through the standard memory interface rather than I/O requests.

This dissertation investigates rewriting the storage stack to account for SCM. The central hypothesis is that direct access to SCM from user mode can form the basis for a flexible high-performance storage architecture. Direct access promises much higher performance by avoiding the cost of entering the kernel and removing layers of code. Direct access also enables flexibility by letting applications customize the storage system to their needs without

extending the kernel. Previous work has shown that matching application needs to storage-system design can deliver major performance improvements. For example, Google's GFS optimizes for web data [69], Facebook's Haystack optimizes for images [19], and the Vertica column-oriented store optimizes for data analytics [109, 176]. These systems provide better performance by specializing for a narrow range of application workloads (*e.g.*, whole file access).

The dissertation presents the design, implementation, and evaluation of an operating-system storage architecture based on direct access to SCM. The architecture comprises two parts. The first, Mnemosyne [188], is a lightweight system for exposing *persistent memory* to user-mode programs. Persistent memory enables programs directly store and access common in-memory data structures, such as trees, logs, and hash tables, in regions of SCM (private to each program). While persistent memory enables low-latency flexible storage, it sacrifices many of the features that make file systems useful such as organizing data under a global logical namespace for easy access, and protecting data for secure sharing between applications. The second part, Aerie, is a flexible file-system architecture that fills this gap. Aerie exposes a file-system interface to SCM using user-mode libraries that access file-system data directly through memory. Aerie retains both the benefits of direct access and the sharing and protection features of file systems.

1.1 Motivation

Emerging device technologies enable low-latency storage, which can greatly benefit modern applications. However, existing operating systems do not expose the full potential of these technologies to applications, thus calling for rethinking the design of operating-system storage architecture and interfaces to exploit new storage technologies.

1.1.1 Technology Trends

Storage-class memory (SCM) [65] technologies promise to change many assumptions about storage. They have the persistence of storage, but the interface of memory, that is they can

be attached to the memory bus and accessed through load and store instructions. Recent technologies that provide SCM capabilities include: phase-change memory (PCM) [115], spin-transfer-torque magneto-resistive RAM (STT-MRAM) [93], and memristors [177]. Although the performance and reliability details differ, they all provide byte-granularity access near the speed of DRAM and the ability to store data persistently across reboots without battery backing. Compared to flash technology, SCM is between two to three orders of magnitude faster and enables finer-grain access for flexible data structures. While SCMs are currently only available in small sizes and scaling projections have flattened in recent years, recent work indicates that even current SCM devices offer great potential for improved storage performance [9].

1.1.2 Application Trends

We see an increasing body of evidence that modern applications could benefit from low-latency storage. These range from desktop and smartphone applications to large-scale web applications.

Modern desktop and smartphone applications have become less comfortable with simply writing data and hoping data is eventually flushed to stable storage. Instead they often explicitly force writes to stable storage to ensure durability [86, 101]. However, frequent flushes to slow stable storage may cause sluggish performance that results in poor user experience. For example, Firefox 3 had a problem with calling `fsync` too frequently [44], bringing the system to a crawl by frequent flushes to disk. Firefox developers found themselves in the uncomfortable position of trading data integrity for better performance by reducing the frequency of syncs. Similarly, a recent study found that the performance of smartphone applications that perform frequent syncs is quite sensitive to storage latency [101]. As a result programmers often devote a good portion of their time to managing and optimizing I/O performance through complex techniques such as batching I/O requests and performing asynchronous I/O to overlap long storage latency with computation [86]. Since such techniques put a huge burden on the programmer, previous work has explored techniques that hide synchronous I/O latency

without modifying the application, such as prefetching [147, 79] and speculation [138, 139].

Web applications, such as search, social networks, and electronic commerce, also require low-latency access to storage. Such applications have limited locality as they make complex unpredictable explorations of data over large datasets in response to multiple different-user queries. Each individual query may comprise multiple dependent lookups that are processed sequentially and therefore contribute to the total query latency. Since user queries must be served in real time, these applications often resort to keeping a large fraction of their data in memory for low-latency access. For example, Facebook keeps approximately 75% of its total data size in memory [141] and Bing allocates much of its index in memory [108].

Finally, other applications include distributed agreement protocols such as Paxos that needs to synchronously write to a persistent log [31], and high-frequency trading that requires fast processing of financial data to perform real-time decisions [166].

1.1.3 Problem

Despite rapid advancements in storage technology, the fundamental architecture of storage in operating systems has remained stable: applications invoke the kernel to store and retrieve data, which invokes a file system and then a block driver. Microkernels and user-mode file systems move the file-system logic to user mode, but leave the remaining layers intact [124, 175]. Databases may bypass the file system, but still call through the kernel and block driver. Four features of past storage technologies require this layered design in the kernel:

1. *Protection:* Disks and other block storage devices do not implement a protection mechanism to limit access by a user or process. In addition, they use DMA to read/write data, which does not respect memory protection. Thus, the OS relies on the file system to decide which processes have access to which blocks on disk.

2. *Scheduling:* Disks have variable latency from seek and rotational delays and benefit significantly from scheduling to reorder requests.

3. *Caching:* Slow disks benefit from shared caches that allow processes to re-use data fetched by another process.

4. *Drivers:* Disks implement a variety of interfaces and therefore require a driver to present a standard block interface.

Due to the slow speed of disks and the high benefits of scheduling and caching, a kernel implementation of file systems provides many performance benefits at little additional cost.

SCM has none of the features that require the operating system to abstract and mediate access to storage through a kernel-mode file system and a block-oriented interface. As memory, it can be protected by existing memory-translation hardware. Furthermore, it has much less need for scheduling to optimize latency, as there are no long seek or rotation delays. Because SCM provide speeds near DRAM, caching data may be unnecessary. Finally, SCM does not require a driver for data access as it can implement a standard load/store or protected DMA interface [30].

Moreover, the existing operating-system storage architecture limits flexibility by imposing a specific persistence model and handicaps performance of SCM by introducing unnecessary overhead costs.

Flexibility. Major performance improvements can come from matching application needs to storage-system design. However, implementing application-specific persistence today entails three unattractive choices. A first choice is to implement a custom kernel-mode file system, which is difficult and requires long-term maintenance as the kernel evolves. User-mode frameworks such as FUSE [175] simplify file system development but decrease performance through extra context switches to a file-system process. Moreover, although both kernel-mode and user-mode implementations allow flexibility in layout, they restrict access through a standard file system interface. A second choice is to layer persistence above an existing file system or database. For example, Microsoft Office file-formats implement a file system within a file [86]. However, such an approach requires extra code to format and access data, which adds additional latency to file access. A third choice is to completely bypass the file system

and perform direct I/O to the block device similarly to modern databases. While this choice works well when a single application accesses the device, it complicates shared access by multiple applications that do not trust each other because: (i) access control is enforced at the granularity of the device, thus making it hard to selectively share individual blocks, and (ii) the device implements no concurrency control so applications cannot coordinate concurrent access to individual blocks in a safe manner. So to support shared access, applications typically perform their I/O requests via a server process with a proprietary interface.

Performance. The existing storage architecture handicaps performance of SCM. For example, while PCM read latencies can be as low as 70 nanoseconds, the time for a `read` system call in Linux on a modern processor is approximately 1200 nanoseconds. There are two major performance impediments with the existing architecture: (i) the architecture requires entering the kernel, which introduces a cost for changing privilege mode and causes cache pollution [42, 173]; and (ii) it introduces unnecessary layers of code that cannot be completely bypassed.

1.2 Thesis and Contributions

Our thesis is that user-mode code should have direct access to storage-class memory. This can provide two benefits: flexibility and performance. First, direct access enables flexibility by letting an application implement persistence customized to its needs without modifying the kernel and without any performance penalty from executing outside the kernel. Second, direct access promises much higher performance by avoiding the cost of entering the kernel and by removing layers of code.

To this end, we built a new operating-system storage architecture based on the idea of direct access. The architecture comprises two main parts, Mnemosyne and Aerie, both which we describe next.

1.2.1 Mnemosyne: Lightweight persistent memory

Our first proposal is that operating systems should expose SCM as a *persistent memory* abstraction to provide direct access to the durability of SCM technologies. This abstraction enables programmers to make in-memory data structures persistent without first converting them to a serialized format. Thus, trees, lists, and hashes can survive program and system failures. Furthermore, direct access reduces latency to storage because it bypasses many software layers, including system calls, file systems, and device drivers.

We built Mnemosyne[1], a lightweight system for exposing persistent memory to user-mode programs. We designed Mnemosyne to run on conventional processors, requiring no special support beyond the necessary memory controller for SCM. A design compatible with existing commodity processors enables the adoption of SCM without the processor support required by other proposals [42, 41], and avoids the chicken-and-egg problem of which comes first, SCM or processor support.

Mnemosyne provides a low-level programming interface, similar to C, for accessing persistent memory. Specifically, it provides three key services that simplify programmer use of persistence. First, Mnemosyne provides *persistent memory regions*, which are segments of virtual memory stored in SCM rather than volatile memory. Regions can be created automatically to hold variables labeled with the keyword `pstatic` or allocated dynamically. Mnemosyne virtualizes persistent regions by swapping SCM pages to a backing file. Second, Mnemosyne provides *persistence primitives*, which are low-level operations that support consistent updates to data in SCM. These primitives enable programmers move data structures between consistent states, automatically recovering to such a state after a failure. Third, Mnemosyne provides a *durable memory transaction* mechanism that enables consistent in-place updates of arbitrary data structures.

We implement Mnemosyne as (i) a pair of libraries that implement persistent regions, persistence primitives and the transaction runtime system, (ii) a compiler that supports persistent variables and transactions, and (iii) a small set of modifications to the Linux kernel

[1]Mnemosyne is the personification of memory in Greek mythology, and is pronounced *nee–moss–see–nee*.

for allocating and virtualizing SCM pages. We implement the persistence primitives using regular x86 instructions with a performance emulator for SCM accesses.

In the evaluation, we show that Mnemosyne provides a simple abstraction for programmers to make data structures persistent. We compare Mnemosyne performance against Berkeley DB running on a RAM disk with the performance of phase-change memory. For small data sizes, Mnemosyne transactions perform 6–14 times better than Berkeley DB. We also convert three applications, OpenLDAP, Tokyo Cabinet and Memcached, to use persistent memory and find the performance of moving existing in-memory data structures to persistent memory is 35–1400 percent faster than Berkeley DB's optimized storage or flushing the whole structure to a file. These results demonstrate that low-latency storage can help improve single-machine application performance for write-intensive workloads, which in turn can benefit large-scale applications based on these services.

1.2.2 Aerie: Application-specific file systems

Although persistent memory enables low-latency flexible storage, it sacrifices many of the features that make file systems useful such as organizing data under a global logical namespace for easy access, and protecting data for secure sharing between applications. We would like to support these features but also retain the flexibility and performance benefits of direct access.

Our second proposal is to distribute file-system functionality for direct and protected access to SCM. Direct access provides two key benefits: (i) low-latency access to data by removing layers of code, and (ii) flexibility by enabling an application to implement a file system customized to its needs without extending the kernel. With direct access, a program reading a file can locate the file contents and read the data directly, without calling the kernel. The file system can also be customized and optimized along several dimensions. The interface of a file system can be optimized based on the application needs, such as get/put for a mail message store rather than open/read/write/close. The interface can also provide concurrency semantics that better target an application's concurrency model, such as byte-range locking

to enable concurrent access to the same file. Moreover, an application can choose a layout suited to its workload, such as contiguous allocation for fixed-size files.

Based on this idea, we designed the Aerie architecture to expose file-system data stored in SCM directly to user-mode programs. Applications link to a file-system library that provides local access to data and communicates with a service for coordination. The OS kernel provides only coarse-grained allocation and protection, and most functionality is distributed to client programs. For read-only workloads, applications can access data and metadata through memory with calls to the file-system service only for coarse-grained synchronization. When writing data, applications can modify data directly but must contact the file-system service to update metadata.

We implement two file-system interfaces on the same layout with Aerie: a POSIX-style system and one optimized for small-file access through a put/get interface. Through experiments, we show that the POSIX-style system performs much better than existing user-mode file systems and between only 15% slower than a kernel file system without consistency guarantees and 35% faster on average than ext3. The specialized key-value file-system interface performs up to 86% faster than the fast kernel file system. Thus, distributing file system functionality to client processes allows flexible implementations that dramatically improve performance.

1.3 Dissertation Organization

Chapter 2 provides background on various aspects integral to this dissertation. It presents a primer on storage-class memory (SCM) and discusses previous approaches exploiting SCM in system design. It then reviews material on transactions and persistence techniques, and finishes with an overview of file system architectures.

Chapter 3 presents the design, implementation, and evaluation of Mnemosyne. Mnemosyne provides programmers with direct access to the durability of SCM through the persistent memory abstraction. With persistent memory, programmers can make common in-memory data structures, such as lists and trees, persistent without having to convert them to a serial-

ized format. Data structures can be made persistent at the speed of the device rather than the speed of the operating system.

Chapter 4 presents the design, implementation, and evaluation of Aerie. Aerie is a flexible file-system architecture that exposes SCM to user-mode programs so they can read and write data without kernel interaction but maintain the naming, sharing, and protection features of file systems.

Finally, Chapter 5 summarizes and offers conclusions.

2 BACKGROUND AND RELATED WORK

In this chapter, we provide background on various aspects integral to this dissertation. First, we present a primer on storage-class memory (SCM) and discuss previous approaches exploiting SCM in system design. We then review material on transactions and persistence techniques, and finish with an overview of file system architectures.

2.1 Storage-Class Memory

We use the term *storage-class memory* to refer to technologies that allow persistent storage to be attached to the memory bus and accessed through processor load and store instructions [65, 27]. We do not consider flash as storage-class memory because it is only accessible as a block device through a driver.

2.1.1 Technology

Four emerging technologies are promising implementations of SCM: phase-change memory (PCM) [115], spin-transfer-torque magneto-resistive RAM (STT-MRAM) [145], memristors [177], and battery-free flash-backed DRAM [8, 180]. While the performance, density, and reliability details differ, they all provide byte-granularity access and the ability to store data persistently across reboots without battery backing.

Table 2.1 shows the performance, density, and reliability properties of several SCM technologies together with those of DRAM and flash for comparison. Performance-wise, PCM and memristor are expected to be as fast as DRAM and between two to three orders of magnitude faster than flash, while STT-MRAM is projected to be even faster than DRAM. Reliability also varies between device technologies: PCM and memristor can endure three to five orders of magnitude more writes than flash, while STT-MRAM has endurance similar to that of DRAM. Finally, while SCM is currently only available in small sizes, prototype implementations and scaling projections indicate that within a few years larger sizes (*e.g.*, 1 GB) will become

	Technology	Read (ns)	Write (ns)	Density (μm^2/bit)	Endurance (writes/bit)
Present	DRAM	60	60	0.00380	$> 10^{16}$
	NAND Flash	25,000	200,000–500,000	0.00210	10^4 - 10^5
	PCM [127]	115	120,000	0.173	10^6
Future	PCM [20, 186, 38]	50 – 85	150 – 1,000	0.00250	10^8–10^{12}
	STT-MRAM [145, 91, 183]	2 – 20	2 – 20	0.0102	10^{15}
	Memristor [158, 133]	100	100	0.00320	10^{10}

Table 2.1: Access latency, density, and endurance for various solid-state storage technologies. Prospective characteristics are based on demonstrated prototypes, and scaling projections for the year 2015 [6, 158]. DRAM and flash are provided for baseline comparison.

available. Eventually, it may be economically feasible, and perhaps cheaper, to outfit a system completely with SCM and no DRAM [115, 151, 199].

Phase-Change Memory (PCM)

Phase-change memory (PCM) stores data by relying on the resistivity difference of the crystalline and the amorphous states of chalgogenide alloy. The material can be switched between the two states by first heating it to a high temperature (over 600°C), and then cooling it either slowly into low-resistance crystalline state or rapidly into high-resistance amorphous solid. The crystalline state can be used to store logic ONE while the amorphous state can be used to store logic ZERO.

Micron has announced volume production of 1 GBit PCM chips [128] making PCM the first commercially available SCM technology. Compared to DRAM, currently available PCM is approximately two times slower for read accesses and 2000 times slower for write accesses [127, 128]. However, access latencies for PCM, as demonstrated by research prototypes [20, 186, 38], are expected to match those of DRAM for reads and be 2-17 times slower for writes.

Moreover, it is predicted that PCM may achieve higher storage density than that of DRAM and flash. Samsung has recently demonstrated an 8 GBit prototype at 20 nm process technology [38], while PCM is expected to scale down to 8 nm process technology [6]. In contrast, scaling projections suggest that flash and DRAM will have difficulties scaling beyond 20 nm [6].

Spin-Transfer Torque Magneto-resistive RAM (STT-MRAM)

Magneto-resistive RAM (MRAM) stores data by relying on the resistivity from magnetization of a magnetic tunnel junction (MTJ), which is a component consisting of two ferromagnets separated by a thin insulator. If the magnetizations of the two layers are in parallel orientation the MTJ has lower resistance than if the magnetizations are in anti-parallel orientation. The difference in resistance can be used to store data, using low resistance to store logic ONE and high resistance to store logic ZERO.

Spin-torque transfer MRAM (STT-MRAM) represents a recent, promising advancement in the MRAM device family [64]. STT-MRAM relies on the spin-transfer-torque effect, where a spin-polarized current transfers its angular momentum to a magnetic layer to switch the magnetic orientation of one of the two layers and therefore effectively change the resistance of the MTJ. Access latencies for STT-MRAM, as demonstrated by research prototypes [145, 91, 183] are expected to beat those of DRAM by 3-30 times, with read speeds as fast as SRAM [81]. Everspin has recently announced commercial availability of 64MBit STT-MRAM chips [60].

Memristors

Memristor stores data by relying on the memristance property, which was theoretically formulated by Chua in 1971 [40] but only recently demonstrated by Strukov *et al.* on nanonscale thin film devices [177]. When current flows in one direction through the device, the resistance increases, and when the current flows in the opposite direction, the resistance decreases. When the current is stopped, the device retains the last resistance it had, and when the current starts again, the resistance of the device will be what it was when it was last active. Low and high resistance levels can be used to store logic ONE and ZERO respectively. Memristors offer access latency comparable to DRAM. HP is planning to release memristor devices by 2014 [122].

Usage	Approach
Main memory	[115, 151, 199, 96, 150, 170]
Fault tolerance	[54, 163, 136, 43, 16]
File system	**Aerie**, [90, 58, 130, 189, 42, 98, 195, 30]
Persistent heap	**Mnemosyne**, [120, 41, 133]
Persistent data structures	[185, 35, 67]

Table 2.2: Approaches to using SCM in system design. Bolded approaches are presented in this thesis.

Battery-free Flash-backed DRAM.

Flash-backed DRAM stores data by leveraging the speed and endurance of DRAM with the non-volatile storage retention of flash. It achieves this by seamlessly integrating three commodity components, DRAM, super-capacitor, and NAND flash into a single module. At normal operation, the storage module is visible to the host environment and functions as a standard DRAM module. When power failure happens, the super-capacitor has enough energy so that the module can save data stored in the DRAM to the NAND flash. When power is restored, the module writes data from the flash back to the DRAM.

AgigaTech and Viking technology announced plans to offer memory modules as large as 8GB [8, 180]. Since this type of SCM is built using ordinary DRAM and flash, its storage density is essentially limited by the least dense technology of the two (DRAM). Thus, its storage scalability is not expected to match that of other types of SCM. So flash-backed DRAM can be seen as an interim SCM solution until the aforementioned technologies mature.

2.1.2 System Design

Table 2.2 lists previous approaches to using storage-class memory (initially cast as non-volatile RAM) in system design together with the techniques presented in this thesis. We next elaborate on these approaches and how they relate to this thesis.

Main-memory Systems

PCM's higher scalability over DRAM has motivated computer architects to study PCM as a DRAM replacement in general-purpose main-memory systems [115, 151, 199], addressing issues related to PCM's longer access latencies, higher dynamic power, and limited endurance compared to DRAM.

Follow up work in the area explored solutions to further address PCM's limited endurance [96, 150, 170]. These solutions change the mapping of physical address to locations in PCM to level writes across more locations, and they could be potentially extended to address similar issues that arise when using PCM for storage.

Fault-tolerant Systems

Previous work has explored using storage-class memory to tolerate and recover from failures, including checkpointing techniques transparent to the programmer but also programmer-driven recovery approaches.

Checkpoint-restart is a well-known technique for tolerating failures due to transient errors: when a failure happens, the system seamlessly recovers and continues operation from a previously taken checkpoint of system state. However, as hardware-failure rates increase due to technology scaling [25], frequent checkpointing is necessary for the technique to be effective. Previous work has explored using SCM to reduce checkpointing overheads and enable frequent checkpointing in massively parallel processing systems [54] and shared-memory multi-core systems [163].

Whole-system persistence (WSP) [136] proposes checkpointing system state on a power failure. When failure happens, WSP relies on a small residual energy window provided by the system power supply to flush any volatile state kept in the processor cache and buffers out to SCM. Upon recovery, all state, both OS and application state, is restored transparently from SCM. WSP, however, cannot tolerate hardware or other catastrophic failures that prevent taking post-failure actions.

The recovery box [16], Safe RAM [43], and the Rio file cache [34] looked at providing a

region of non-volatile memory that survives failures and can facilitate recovery. These projects differ on the intended use of the non-volatile memory region and the interface provided to the programmer. With the recovery box, programmers can store operating system or application state that they can later use to speed up recovery. With Safe RAM, programmers can designate memory pages that contain reliable updates such as data or log pages in a database system. This enables programmers to update such pages without having to force updates to disk for durability, which can improve performance. Finally, the Rio file cache [34] keeps the file-system page cache in non-volatile memory so that file data can survive a power loss without having to force them to disk for durability. Rio also uses hardware virtual memory protection to protect the cache against operating system errors, such as wild stores, that may corrupt the cache during a system crash.

File Systems

There have been several prior projects investigating the integration of storage-class memory into file systems. Initially cast as non-volatile RAM (NVRAM), these memories can be used as persistent write buffers to reduce the latency of writing data [90], store just file system metadata [58], or hold frequently changing metadata and small files [130, 189].

More recently, the BPFS file system leverages SCM's byte addressability to reduce the amount of metadata written during an update and relies on special processor support for ordering to achieve consistency efficiently through shadow updates [42]. FRASH keeps page metadata in SCM to improve performance [98], and SCMFS utilizes the existing memory management module in the operating system to remove block management from the file system [195].

Those approaches all improve performance for file access but the fundamental file system and storage architecture are left unchanged. They provide no flexibility in file system interface and organization. So applications cannot customize and optimize the file system to their workload, which can be beneficial to applications as we demonstrate with Aerie (Chapter 4). Moreover, the file system code runs in kernel mode, which introduces a cost of changing

modes and cache pollution from entering the kernel [42, 173]. A user-mode file system can avoid these performance costs and achieve lower latency and higher bandwidth than going through the kernel. The Moneta-D system makes a first step towards user-mode access by bypassing the kernel for data access [30], but metadata operations critical to small-file performance, such as for IMAP servers that use many small files [55], still call into the kernel. Aerie can access file-system metadata from user mode, which can benefit workloads that operate on many small files.

Persistent Heaps

Close to our work on Mnemosyne (Chapter 3) is past and recent work on persistent heaps [120, 41, 133].

Rio Vista [120] is a lightweight, user-mode, recoverable-memory library that runs on top of the Rio file cache. Similarly to Mnemosyne, the Vista recoverable memory provides atomic updates and persistence for a region of virtual memory. Vista also provides a persistent heap that helps programmers manage memory within a persistent region of virtual memory. Vista relies on the Rio file cache to automatically persist a virtual memory region on a system crash or power failure, which simplifies Vista's crash-consistency code. In contrast, Mnemosyne takes no post-failure actions. Instead, to ensure that any hardware volatile state, such as state kept in the processor cache, is not lost on failure, it properly flushes updates to SCM in a crash-consistent order.

NV-heaps [41] is a concurrently developed project that similarly to Mnemosyne exposes SCM to programmers through a persistent heap and provides transaction updates to data. However, the focus of the two projects is different: NV-heaps focus on persisting user-defined *objects*, so they support transactions via an object-based transactional memory system. In contrast, Mnemosyne supports transactions via a word-based transactional memory system, which can be used to transactionally update any data kept in persistent memory. NV-heaps provide some features not currently found in Mnemosyne, such as type-safe pointers and garbage collection of persistent objects via reference counting. However, since NV-heaps is

designed for an unmanaged language such as C++, it is not totally safe as it still suffers from accidental errors that corrupt memory, such as wild stores. Finally, while Mnemosyne can be used with existing commodity processors, NV-heaps require special processor support for ordering.

Moraru *et al.* [133] focus instead on reliability techniques for SCM-based persistent heaps, making their work orthogonal to our work. They built a heap that uses checksums to detect and recover from metadata corruption, which they complement with a mechanism based on virtual-memory protection for containing erroneous writes.

Persistent Data Structures

Several research studies have examined the redesign of data stores and data structures in light of SCM. Venkataraman *et al.* [185] proposed a methodology for building consistent and durable data structures for SCM, which they followed to build a SCM-based B-tree for use as a key-value store. Chen *et al.* [35] looked at designing efficient database algorithms such as B^+-trees and hash-joins that reduce execution time and energy when running on PCM while increasing write endurance. Finally, Gao *et al.* [67] proposed a design of a logging scheme that exploits PCM for transaction logging in disk-based database systems.

2.2 Transactions

Mnemosyne (Chapter 3) provides programmers with durable memory transactions to consistently update data structures in the presence of failures. We next review background material and related work on transaction processing and memory transactions.

2.2.1 Transaction Processing

The use of transactions for grouping a set of operations into a single unit of work that appears to execute indivisibly and instantaneously has been the focus of the database and systems research community for almost three decades [75, 76, 131]. Transactions have been appealing primarily because they free the application or system programmer from dealing with two

complex issues: (1) *concurrency*, that is all the effects arising from concurrent execution, and (2) *failures*, that is all the effects caused by unexpected interruptions of program execution due to software or hardware failures.

A transaction-processing system should guarantee four properties when executing transactions, abbreviated by the acronym ACID [77]:

1. *Atomicity.* All operations of the transaction complete successfully or none of these operations appears to execute. A transaction that completes successfully *commits*, otherwise it *aborts*.

2. *Consistency.* A transaction transforms data from one state that is consistent according to some application defined integrity rules to another state consistent to those rules. For example, in a course reservation system a rule might require that a student cannot register for a course if the current number of registered students equals the maximum allowed.

3. *Isolation.* This property dictates the behavior of concurrent transactions that act on the same data. The most strict behavior is *serializability*, which dictates that transactions appear to execute sequentially. In such a case, intermediate states of data are not visible to other transactions.

4. *Durability.* Once a transaction commits, all its effects must be preserved, even if there is a failure.

A transaction system relies on two main components to support transactions: (1) *concurrency control*, which guarantees isolation, and (2) *recovery*, which guarantees atomicity and durability. A classic approach to implementing transactions uses locks for concurrency control [12], and write-ahead logging (WAL) for recovery [119, 74].

WAL writes all modifications to a stable log before these are applied to the stable image to allow completing the execution of any transaction interrupted by a failure. However, buffer management employed by disk-based database systems to improve performance complicates

WAL. WAL protocols may not flush data updates to the disk on each transaction commit, but delay flushing until the buffer manager replaces the corresponding data page [131]. This optimization enables flexibility in the replacement policy of the buffer manager and removes random I/O due to data flushes from the commit path. In contrast to disk-based databases, main-memory databases [68, 51] store all data in memory rather than caching disk-data, but they still rely on a disk for persistently storing a log of committed updates. Finally, WAL relies on the ability to write entries to the log atomically. However, in practice a write to a disk page is not atomic so transaction systems must deal with partial writes, which they often solve with checksums [129].

Recent work by Prabhakaran *et al.* [149] and Ouyang *et al.* [144] has investigated supporting the atomicity and durability properties of transactions directly in flash-based storage devices. The idea is to export a *atomic-write* transactional interface to higher-level software that allows an application to group multiple writes to flash as a single transaction. Ouyang *et al.* [144] use a single-bit-per-block tracking technique similar to our torn-bit log (Section 3.2.4) to identify contiguous flash blocks that are part of the same transaction. In contrast to our torn-bit log, their technique does not have to deal with torn writes that happen when later writes complete before earlier ones do.

2.2.2 Memory Transactions

Transactional memory (TM) [87, 84] focuses on the concurrency of transactions rather than on durability with the aim of providing a simple mechanism for synchronizing concurrent reads and writes of shared data in a concurrent program. Specifically, transactional memory lets a programmer to declare a block of code as an *atomic region* with the `atomic` keyword. This simple yet expressive high-level language construct allows the programmer to express mutual exclusion without having to resort to other low-level and error-prone synchronization mechanisms such as mutex locks.

The TM system executes atomic regions using memory transactions that guarantee atomicity (i.e., all or nothing) and isolation (i.e, no access to uncommitted data). It provides atomicity

by either logging old values in an undo log or buffering new values in a redo log to allow the transaction to abort. It enforces isolation by detecting when two transactions conflict, which happens when two concurrent transactions access the same memory locations and one transaction performs a write. Memory locations are memory words in a word-based TM system and objects in object-based TM systems. When a conflict occurs, a resolution policy may stall or abort one of the transactions to clear up the conflict. Transactions can execute concurrently if they do not conflict. Thus, they can improve performance if critical sections rarely conflict.

Both software and hardware implementations have been proposed. Software TM implementations instrument code to record the locations read and written and detect conflicts either at commit or while the transaction executes [85, 52, 161, 62, 56, 48, 84]. As a result, software TM implementations may slow down critical sections by 3-5x [48]. Proposed hardware TM implementations, in contrast, perform these operations in hardware with less slowdown to critical-section code [82, 152, 132]. However, feasible hardware TM implementations often bound the number of distinct memory locations that can be accessed within a transaction. Thus, they are best paired with a software TM for fallback when transactions exceed hardware limits [49, 47].

Transactional memory is reaching a maturity level where it can be used in deployed applications. On the software side, Intel and GCC compilers provide extensions for using transactional memory [4, 88]. On the hardware side, Intel announced support for TM in their upcoming Haswell micro-architecture [94], IBM has integrated TM into their Blue Gene/Q supercomputer processor [83], and AMD has indicated it may implement limited TM systems in hardware [11].

2.2.3 Durable memory transactions

Previous work on durable memory transactions focuses on the atomicity and durability properties of transactions. Transactions are used to atomically update persistent data kept in recoverable memory, which is a region of memory that survives crashes.

Lightweight recoverable virtual memory (RVM) [164] supports durable memory transactions by committing data at page granularity, which may require up to three copies: (1) when updating a memory location, it copies the before-image to an in-memory undo log to support user-initiated aborts, (2) when the transaction commits, it writes the updated data to an on-disk redo log to support atomicity and durability, and (3) when the on-disk redo log fills up, it truncates the log by propagating updates to the on-disk database. Rio Vista [120] improves the performance of this design by relying on the Rio file cache to avoid synchronous writes to the disk.

Several other projects have also supported page-wise consistency mechanisms for atomicity. Several IBM systems have provided transactions in memory by marking pages accessed by a transaction in a TLB-like structure [32, 174]. Similar to RVM, this approach provides coarse-grained transactions at the page granularity, and relies on disk for durability. QuickStore [191] also implements memory transactions at the page granularity, and Texas [172] provides a page-wise checkpointing mechanism for consistency. We elaborate on Quickstore and Texas in Section 2.3.

Mnemosyne, our proposal, is similar to RVM and Rio Vista. Mnemosyne borrows the notion of persistent regions from RVM, but commits data at word granularity rather than at page granularity. Unlike Rio Vista, Mnemosyne transactions operate completely at user level and do not require a battery back-up system. Mnemosyne also relies on software transactional memory techniques to automatically annotate program code in a compiler, and to implement efficient commit processing. As a result, our system benefits from recent research in the area [194].

Finally, compared to database systems and persistent stores (Section 2.3), Mnemosyne focuses on latency of individual memory transactions rather than transaction throughput. Thus, Mnemosyne leverages the low-latency of SCM to avoid incorporating techniques that improve throughput at the expense of latency such as group commit where multiple transactions with commit records on the same log page are committed as a group [51].

2.3 Persistence Techniques

Mnemosyne (Chapter 3) abstracts storage as persistent memory; programmers can create a single data structure, optimized for memory, rather than designating separate in-memory and update-optimized persistent structures. Persistent memory provides a flexible interface to storage, thus enabling seamless integration of persistence into the application. We next discuss previous work on interfaces and abstractions for persistence, which we present in increasing level of flexibility.

Data-storage Libraries

Data-storage libraries represent a popular persistence solution among developers with many available frameworks. Such frameworks range from rigid embedded databases, which provide a database interface to storage, to extensible storage frameworks, which enable application-specific storage.

An embedded database refers to a database that does not run in a separate process but instead is available as a library that is directly linked into the application that requires access to the stored data. The database though is not integrated into the language so the programmer has to transform the application data represented in the language's data model (*e.g.*, objects) into data represented in the database's data model (*e.g.*, relational or key-value tables). This is contrast to Mnemosyne, where there is a single representation of the data and the programmer is not burdened with maintaining code to transform data between multiple data representations. Popular embedded databases include SQLite [3] and BerkeleyDB [140, 46]. SQLite provides a transactional SQL database engine. BerkeleyDB provides a transactional key-value store offering hash and B+tree access methods.

Anvil [123] is a modular and extensible toolkit for building database backends. The basic Anvil abstraction is an abstract key-value store. Anvil provides several concrete key-value store implementations, each optimized for different access pattern. Programmers have flexibility to combine them in multiple ways to build a key-value store that matches an application workload. Since storage is always accessed through a key-value interface, programmers

still have to write code that transforms an application's in-memory data structures into the key-value data model.

Stasis [168] is an extensible storage framework that generalizes write-ahead logging algorithms to provide applications with flexible transactional storage. In contrast to databases, Stasis does not restrict the data model and access methods but instead provides programmers with primitives for constructing highly concurrent application-specific data structures. Unlike Mnemosyne, Stasis does not provide a compiler for automatically constructing transactions to consistently update a persistent data structure.

Serialization Frameworks

Serialization frameworks provide a mechanism for serializing and deserializing objects, and can be used to persist objects. Similarly to Mnemosyne, programmers maintain a single in-memory version of an object (or data structure). With serialization, they can automatically convert the in-memory structure into a sequence of bytes, which they can then store in a storage medium. Then, with deserialization they can automatically reconstruct an object out of its byte sequence and load it into memory. Serialization frameworks can either be loosely integrated into the language through syntactic sugar such as the `Serializable` attribute in C# or come in a library form such as the C++ Boost Serialization and the Java Serialization API.

Serialization may perform *pointer unswizzling* [134] to convert a memory-pointer into a position-independent reference such as referencing an object using its unique object identifier (OID). This decouples an object from its memory location and allows reloading the object into a different memory address upon deserialization. Deserialization then performs pointer swizzling, which replaces position-independent inter-object references with the main-memory address of the referenced object so as to speed up access along such references. Mnemosyne persists data directly so it avoids the cost of pointer swizzling and unswizzling.

Language integrated persistent stores

Previous research on persistent object systems attempted to provide language level persistence through the integration of database systems and programming languages so as to relief the programmer from managing persistence. Such language frameworks could be layered on top of Mnemosyne, to provide higher-level services such as garbage collection and safe references to ensure that persistent data does not point to volatile data.

Atkinson *et al.* [14] introduced the seminal PS-Algol persistent language, which extended Algol with functions to store objects in a database system. PS-Algol introduced the concept of orthogonal persistence, that is persistence for all data irrespective of their type, and transitive persistence, that is persistence for all data reachable from a root. Mnemosyne persists data directly so it similarly provides persistence for all data irrespective of their type. Also, since Mnemosyne persists data directly for lower latency, it exposes to the programmer a strict separation of persistent and volatile memory where persistent memory may reach to volatile without making it persistent, and vice versa. Thus, in constrast to PS-Algol, Mnemosyne does not provide transitive persistence and leaves the programmer responsible to avoid pitfalls such as persistent memory leaks. Finally, PS-Algol did not provide any mechanism for consistency.

Other important projects that followed up include Alltalk [156] based on Smalltalk, the E language [155] that integrated persistence into C++ and brought other features beyond persistence such as iterators for structuring database queries, SHORE [29] that provided a persistent object store accessible via C++ bindings, and PJava [15] and PJama [13] based on Java. SHORE addressed some of the weaknesses of the E language and its underlying object store. It supports typed persistent objects by storing type information along with each object and describing types in a language-neutral notation. It also provides a rich naming environment by making the object namespace available through the UNIX file system namespace, a feature that facilitated locating persistent objects outside the application. PJava and PJama incorporated the concepts of orthogonal and transitive persistence into the Java language. However, to achieve this, they required extending the Java virtual machine (JVM), a property that prevented their adoption into a mainstream JVM.

Virtual memory-based persistent object systems, such as ObjectStore [110], Texas [172] and QuickStore [191] rely on pointer-swizzling at page-fault time and virtual memory mapping techniques to trigger the transfer of persistent objects from the persistent store to main memory. This allows them to handle large datasets without having to load the whole persistent store into memory at once. While these systems offer persistence transparent to the programmer, they still rely on a C++ object-oriented programming model, which limits the applicability of the system to only OO programs. Mnemosyne provides greater flexibility by not requiring C++ objects. Moreover, these systems operate on virtual memory pages, which makes the system heavier weight, as they must read or write more data at a time and rely on optimizations such as page-diffing to reduce the amount of data written. Mnemosyne instead persists data directly at the granularity of a single update rather than a whole page, which enables providing lower latency. Finally, both systems implement page-wise consistency mechanisms for atomicity, which unnecessary read and write more data than was requested by the application. Objectstore implements transactions using the write-ahead log protocol: updated pages are written to the log and only propagated to the database when the transaction commits. Texas provides a page-wise checkpoint mechanism for consistency where every page modified after a checkpoint is written out to a log as part of the next checkpoint. This checkpointing mechanism, however, is insufficient in the presence of multiple threads as it requires quiescing threads before taking the checkpoint.

The higher-level interface to these systems and others, such as Thor [118], allows them to provide safety properties, such as only allowing pointers in persistent data structures to reference other persistent data (or, in the case of Thor, to make persistent any volatile structures referenced). Their knowledge of all pointers in a data structure also enables garbage collection to prevent memory leaks.

Persistent Memory

Persistent memory is an old idea whose time has come. An early implementation of persistent memory in the form of recoverable virtual memory, that is regions of virtual address space

that survive a restart, was demonstrated by Thatte [181]. Thatte implemented RVM through a checkpointing scheme that incrementally captured the entire state of the machine. Lightweight recoverable virtual memory [164] provides regions of virtual address space on which transactional guarantees are offered through durable memory transactions (Section 2.2.3). Compared to ordinary memory-mapped files, recoverable virtual memory provides stronger consistency guarantees. Most recently, Guerra *et al.* proposed software persistent memory (SoftPM) [80], where programmers define persistent roots, and SoftPM atomically persists all data reachable from a root when the programmer requests a persistent point. Overall, in contrast to Mnemosyne that persists data at word granularity for lower latency, these systems persist data at page granularity, thus requiring a complete page write for updates that are finer grain than a page.

Several other projects have investigated the use of a large battery-backed memory for persistent storage [26, 43, 59]. However, the memory in these systems is accessible only through a driver to prevent corruption, and hence only available through a database or file system. The eNVy system [193] focuses on the architecture of attaching flash to the memory bus as persistent memory, and uses a battery-backed SRAM buffer to hide the block-addressable nature of flash.

Global Persistence

Global persistence is the embodiment of persistence in its purest form. In contrast to Mnemosyne and other persistence techniques where the programmer designates what is persistent, global persistence is transparent to the programmer: applications use only in-memory objects that survive restarts, and do not distinguish between persistent and volatile objects. This helps programmers avoid pitfalls arising from mixing persistent and volatile objects such as dangling persistent pointers to volatile objects.

Atlas pioneered the concept of single-level store [100] where the application does not distinguish between main-memory and disk storage but instead has a single memory view of the entire storage. Two capability-based operating systems, KeyKOS [113] and Grasshop-

per [159], implemented system-wide orthogonal persistence through periodic checkpointing to disk. To the KeyKOS and Grasshoppers applications, similarly to the single-level store concept, all data lives in a single data-abstraction type called *domain* in KeyKOS and *container* in Grasshopper. In addition, KeyKOS and Grasshopper define a programming model for long-running processes.

Whole-system persistence (WSP) [136], which we discussed earlier in Section 2.1.2, also provides global persistence. Compared to KeyKOS and Grasshopper, WSP requires less or no OS support and targets systems based on SCM. WSP though does not provide a programming model for long-running processes that KeyKOS and Grasshopper provide. Compared to Mnemosyne, WSP lets applications benefit from SCM's persistence without having to be modified to store objects in a persistent heap. Similarly to Mnemosyne, WSP still relies on files for exporting and storing data in an external format.

2.4 File System Architecture

We begin with an overview of the file system's role in abstracting storage devices and enabling protected access to shared data, which is a key motivating factor behind Aerie (Chapter 4). We then discuss architecture features of local and distributed file systems from which Aerie borrows design elements.

2.4.1 Overview

A file system is a software layer that abstracts persistent storage into named collections of bytes called *files*. Traditionally, file systems have been built on top of block-level storage, which is either provided physically by a single storage device (e.g. hard disk) or virtually by an intermediate hardware or software layer that abstracts a pool of physical storage devices (e.g. RAID [146], Petal [116]). Several other projects have explored building file systems on primitives higher than the block abstraction. ZFS [23], object-based storage devices (OSD) [70] and the logical disk [50] all expose groups of blocks and atomic operations. While ZFS collects block storage under a transactional object store, OSDs push the object functionality into the

storage device itself instead. Boxwood [121] explored fault-tolerant data structures such as fault-tolerant B-tree as the fundamental storage infrastructure for building distributed file or other storage systems. Similarly to these projects, Aerie exposes primitive structures through an intermediate storage-abstraction layer to facilitate the construction of file systems.

Files enable users and applications to conveniently and permanently store data for future retrieval and securely share data with other users and applications. In contrast to other methods for sharing data, such as a database, the file abstraction has fewer semantics. For sharing, the file system is concerned with providing a shared namespace, protection, and consistency semantics.

Shared namespace. A file system organizes files into a shared namespace to facilitate locating and sharing files. A common organization found in most file systems is that of a hierarchical namespace. Under this organization, files are grouped into directories, and directories may be placed into other directories to form a tree (or hierarchy) of directories. A file may be placed in multiple directories, in which case the overall namespace structure becomes an acyclic graph. Other namespace organizations are also possible such as a flat namespace where all files are grouped into a single directory or a tagged, search-based namespace where files are tagged with several attributes or keywords and located through search queries on those tags [169]. Aerie is not restricted to a specific namespace organization but leaves this to the file system.

Protection. Protection controls access to files by limiting the type of file access that can be made by each user or process. A common approach to protection is the use of per file permissions in the form of access control lists (ACL), which state the file system operations allowed by each user. The extent the namespace is involved in protecting access to a file varies between systems. For example, POSIX assigns an ACL per file name and enforces namespace permissions along the path to a file, that is it checks the permission of each directory name found in the path. Thus, a file with multiple path names may have different access permissions. Windows similarly assigns an ACL per file name but by default it enforces file permissions

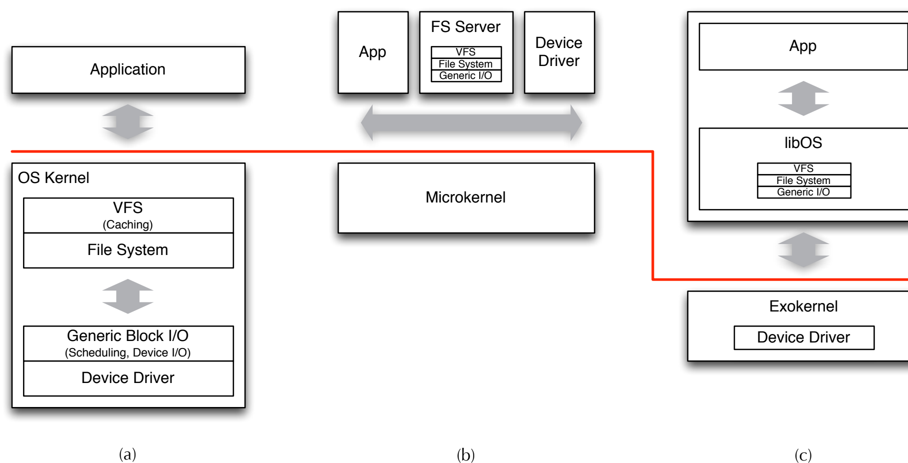


Figure 2.1: Local file-system architectures. The crosscutting red-colored solid line separates mode of operation between user mode (above the line) and kernel mode (below the line).

only and does not enforce namespace permissions. In Aerie, the unit of protection is finer grain than the file, which gives flexibility to the file system in constructing its ACLs and protection control.

Consistency semantics. Consistency semantics deal with the views of shared files, that is when and how modifications by one user (or process) are observed by other users (or processes). POSIX observes *sequential semantics*, that is any file read operation returns the result of the most recent write operation. Although POSIX semantics are easy to support efficiently in a centralized system such as local file system, they put great strain in a distributed setting. As we discuss later (Section 2.4.3) many distributed file systems relax consistency semantics for improved performance. Similarly, Aerie file systems may relax consistency semantics to improve performance.

2.4.2 Local file systems

Aerie delivers a new architecture for local (single-machine) user-mode file systems. Here we review the major architectural features of current local file systems splitting the presentation between kernel-mode and user-mode file systems, and also discuss how Aerie relates to these.

Kernel mode

Figure 2.1a presents the storage stack of modern UNIX-style operating systems including Linux and Solaris. This structure has remained nearly stable over the last four decades resembling that of early UNIX systems [157, 117]. It is designed around the assumptions of past storage technologies such as slow block-based disks despite rapid advancements in storage technology such as solid-state storage. Windows has similar organization.

At the bottom of the storage stack is the disk and other block storage devices. Such storage devices do not implement any protection mechanism to limit access by a user or process. In addition, they use DMA to read/write data, which does not respect memory protection. Thus, the operating system takes sole control of the devices to provide protection. Through the kernel-mode file system, the operating system enforces permission on access to decide which processes have access to which blocks on disk.

Above the storage devices lies the generic block I/O layer. This layer presents a generic block device abstraction to the layers above and implements various optimizations such as scheduling. Since storage devices implement a variety of interfaces, the block layer includes drivers that hide the device intricacies to present a standard block interface. This layer also includes a scheduler, which reorders block requests to minimize total service latency. For example, disks have variable latency from seek and rotational delays and benefit significantly from scheduling that reorders requests in order of physical proximity.

At the top of the storage stack sit the virtual file system (VFS) layer [104] and various file system implementations. VFS provides a common standard interface (*e.g.* POSIX API) between applications and the underlying file systems. Applications invoke VFS through system calls. VFS also serves as a common framework for implementing file systems in the kernel by providing generic implementations of standard services including a common file system namespace, and in-memory caches for frequently accessed file-system data and objects such as directory entries and file inodes (metadata). Caching helps improve performance as subsequent requests can be serviced quickly from memory rather than going to the slow disk. Moreover, shared caching as provided in the kernel allows processes re-use objects

and data fetched by another process. The VFS layer enables file system extensibility and implementation flexibility but restricts access to file systems only through the fixed interface defined by POSIX. Thus, in contrast to Aerie, it does not enable file-system interface flexibility.

Some modern file systems further split the implementation into two layers: (1) an upper layer that interfaces to VFS and implements the file system semantics, and (2) a lower layer that implements the on-disk structures and exposes storage to the upper layer in a form higher than the block abstraction. This division of work enables code reuse and independent evolution of the two layers. For these reasons, Aerie follows a similar layer structure. Examples of modern file systems include ZFS [23], Btrfs [142] and ReFS [45]. A common denominator of these systems is the data integrity features they provide at the lower layer. They all expose a transactional object store implemented using shadow paging to reliably update structures on disk. They also rely on techniques like checksumming and replication of on-disk structures to detect and correct many forms of disk corruption, including lost and misdirected writes, and bit-rot (gradual degradation of data on the media).

Although a kernel-mode file system offers several performance benefits through shared caching and scheduling, its implementation is a long, expensive, and arduous task. This difficulty is both due to the inherent complexity of file systems and the challenging nature of kernel-mode development, which further exacerbates that complexity. As a result, most efforts focus on delivering robust implementations of a few generic file systems that live behind a standard interface such as POSIX. For example, the ext filesystem, Linux's major file system, has had only two major revisions (ext3, ext4) since its introduction about two decades ago. In the same vain, NTFS, Microsoft's main file system for Windows, has had only one major revision in its nearly twenty-year history. ReFS, which is the next generation of NTFS, reuses the upper layer of NTFS. Developers may often use such generic file systems as the foundation for other storage systems implemented in user mode. For example, Facebook's Haystack is an object store optimized for images, which uses a generic UNIX-like file system [19].

User mode

Microkernels push common kernel functionality, including file-system functionality, outside the kernel into user-mode server processes for less complexity and better reliability [197] (see Figure 2.1b). Developing a file system in user mode can be less complex than in kernel mode, resulting in a more robust and reliable implementation. Developers do not have to understand internal kernel APIs and complex locking disciplines. In addition, development in user mode can have a shorter cycle: bugs do not trigger a lengthy crash and reboot cycle, user-mode debugging facilities are generally more powerful than kernel ones, and programmers may use better and safer languages than C. System reliability may also increase because a fault is contained within a process and does not crash the whole system. Clients however may still lose data if the file system does not properly recover. Overall, while a microkernel moves file-system functionality to user-mode, it retains the model of a shared file system that accesses storage through a shared device.

The idea of user-mode file systems has found fertile ground even in monolithic kernel systems. In this context, file-system logic is moved to user-mode, but the remaining layers, such as caching and generic block I/O, remain intact. Moving the logic to user mode introduces extra crossings of the user-kernel boundary, which causes user-mode file systems to perform lower than their kernel-mode counterparts. For this reason, user-mode file systems are mainly used to provide useful features on top of other kernel-mode file systems, such as encryption and versioning, or to expose a file-system interface to non file-system resources such as the Web or a database. Two popular examples of user-mode file-system frameworks are SFS [124] and FUSE [175]. The SFS toolkit facilitates the construction of a user-mode file system as a NFS loopback server. FUSE facilitates the construction of a user-mode file system as a module running in its own user-mode process that responds to callbacks from the VFS layer. Since FUSE is tightly integrated with VFS, it provides no interface flexibility.

Finally, Exokernel [99] and Nemesis [17] have explored exposing storage to user-mode for application performance and flexibility (see Figure 2.1c). However, they still maintained protection of the block device within the kernel, so disk access still required invoking a kernel-

mode device driver. In Exokernel, the kernel exposes physical-storage blocks to clients. When a client creates a new file system, it provides the kernel with functions that the kernel can use to interpret file system metadata. The kernel uses these functions to determine whether a client has access to a block. In Nemesis, the kernel allows user-mode clients setup a secure channel to the storage device, which they use to access the device. The channel ensures a client does not perform I/O to addresses it does not have access permission. Aerie and Moneta-D (discussed earlier in Section 2.1.2) bypass the kernel and access storage directly by relying on virtual memory protection and special hardware, respectively, to enforce file-system permissions.

2.4.3 Distributed file systems

Aerie has been influenced by distributed file-system designs and naturally bears many similarities to them. Features of distributed file systems that are most closely related are consistency semantics and metadata management.

Consistency semantics

Distributed file systems must maintain consistency across multiple replicas of the same file that may arise either due to caching for performance or replication for fault-tolerance. In Aerie, multiple replicas may arise when a client process has local outstanding modifications to a file that has not made visible. Below, we review the consistency semantics of major distributed file systems, and how some of these systems relax semantics for improved performance. Aerie file systems could similarly relax semantics to improve performance.

Several distributed file systems observe sequential semantics at the expense though of performance and implementation complexity. Sprite [143] disables caching when there are conflicting concurrent accesses to a file by multiple clients to simplify cache consistency at the expense of performance. Frangipani [182] and Farsite [7] instead rely on complex distributed lock protocols to maintain consistency in the presence of conflicting accesses. Frangipani acquires per file read or read/write locks to cache and read or write files respectively. Farsite

relies on four different classes of locks (file content locks, namespace locks, shared access locks, and deletion locks) to provide consistency semantics equivalent to those observed on a single local Windows machine.

In contrast, some distributed file systems relax file-system consistency semantics for improved performance. NFS clients cache data blocks for 30 seconds before writing it back to the server [162]. NFS file semantics do not prevent client-cached files from becoming stale as clients can use cached data blocks for 30 seconds without checking with the server. A write to the server may also span multiple requests and hence it may be intermixed with writes from other clients. AFS [92] provides *close-to-open* semantics: writes are pushed to the server once the file is closed, and become visible to any clients that open the file after the close. GFS [69] relaxes semantics for shared atomic appends: appends are guaranteed to happen atomically at least once in the presence of concurrent mutations, thus putting the burden on applications to detect append duplications. Finally, other systems avoid the consistency problem all together by making shared files immutable [178, 171].

Metadata management

Aerie enables file systems with direct access to storage. This design approach is similar in concept to network block stores, such as Frangipani and Petal [116, 182]. These systems abstract a pool of storage devices as a block store on top of which they layer a distributed file system running in a cluster of machines that have direct access to the block store. Machines trust one another so direct access is practical in providing good scalability and performance.

Aerie, in contrast, assumes that processes running the library file system do not trust one another. Managing file-system metadata becomes challenging as Aerie strives to provide the performance benefits of direct access while ensuring untrusted processes do not corrupt metadata. In this respect, Aerie is closer to distributed file systems like Coda [103], Farsite [7] and Ivy [135], which distribute file system functionality to untrusted clients. Similarly to these systems, to ensure untrusted clients do not corrupt metadata, Aerie reintegrates clients' changes to the file system by verifying and replaying operations previously written to a log.

3 MNEMOSYNE: LIGHTWEIGHT PERSISTENT MEMORY

This chapter proposes that operating systems should expose SCM as a *persistent memory* abstraction to provide direct access to the durability of SCM technologies. This abstraction enables programmers to make in-memory data structures persistent without first converting them to a serialized format. Thus, trees, lists, and hashes can survive program and system failures. Furthermore, direct access reduces latency to storage because it bypasses many software layers, including system calls, file systems, and device drivers.

We next present the design, implementation, and evaluation of *Mnemosyne*, which is a lightweight system for exposing persistent memory to user-mode programs. Mnemosyne provides three key services that simplify programmer use of persistence. First, Mnemosyne provides *persistent memory regions*, segments of virtual memory stored in SCM rather than volatile memory. Regions can be created automatically to hold variables labeled with the keyword `pstatic` or allocated dynamically. Mnemosyne virtualizes persistent regions by swapping SCM pages to a backing file. Second, Mnemosyne provides *persistence primitives*, low-level operations that support consistently updating data. Finally, Mnemosyne provides a *durable memory transaction* mechanism that enables consistent in-place updates of arbitrary data structures.

3.1 Design

The purpose of Mnemosyne is to reduce the cost of making data persistent. Current programs devote large bodies of code to formatting data for persistence either in a file system or database, which manage consistency of persistent data. They also carefully decide when to make data persistent, as writing data out to disk frequently leads to poor performance. In contrast, with Mnemosyne, a program should be able to make any data persistent, at any time, with little extra effort.

Mnemosyne presents an abstraction of persistent memory to programmers. We have three goals for the system:

1. *User-mode access* to persistence avoids the latency of entering the kernel and provides flexibility in how persistence is achieved.

2. *Consistent updates* modify persistent data without jeopardizing correctness in the presence of failures.

3. *Conventional hardware* lowers the barrier to adoption by allowing existing processors to work with new memory technologies.

Mnemosyne is designed as a low-level interface to persistent memory, providing necessary methods for consistency. It leaves the separation of volatile and persistent data and prevention of memory leaks to higher levels of software. It also leaves the implementation of reliability mechanisms to address wear out to lower-level software or hardware layers [150].

Mnemosyne achieves the first goal with *persistent regions*, which are segments of memory that are created and virtualized by the kernel and may be accessed directly from user mode. Mnemosyne provides a three-layer solution to the second goal of consistent updates. The upper and middle layers of Mnemosyne provide *persistence primitives*, which are support routines for programming with persistent memory. The lower layer provides hardware primitives for writing data persistently and ordering writes, which is useful for single variable updates. The middle layer provides a persistent heap for allocating small blocks of memory and a log facility to support appends-only updates. The upper layer provides *durable memory transactions* supported by a compiler to enable general-purpose code to create and modify persistent data structures. To satisfy the third goal, our consistent update mechanism relies on hardware primitives available in existing processor architectures. Figure 3.1 shows how the components are divided between user and kernel modes. While we designed Mnemosyne for a system with both DRAM and SCM, it applies equally well to a system that uses SCM for volatile storage as well.

3.1.1 Assumptions

We designed Mnemosyne based on assumptions about the features of SCM hardware and failure models.

Hardware First, we assume like others [42], that SCM is placed directly on the memory bus side-by-side with DRAM, allowing access through normal load/store instructions protected by virtual memory hardware. We believe this is a reasonable and plausible design choice given that computer architects have already studied phase-change memory, a form of SCM, as a DRAM replacement in general-purpose systems [115, 151, 199]. Second, while memory systems for SCM are not yet available, we make several assumptions about the features they provide: (1) we assume they can support an atomic write of at least 64 bits [42, 144], (2) we assume that it is possible to stall execution until a write has made it all the way to SCM, similar to the `fsync` call for file systems, and (3) we assume they implement necessary reliability mechanisms to address wear out. Such reliability mechanisms could be based on previous hardware-based proposals that remap physical addresses to locations in SCM to address wear out when SCM is used as a DRAM replacement [150, 170]. These solutions have low overhead as they rely on a randomized function rather than an indirection table for remapping physical addresses. However, they would have to be extended for persistence so that any state used for remapping survives power loss. Otherwise, wear out could be addressed in software but at the expense of higher performance overhead [133].

Failure Models. While data stored in SCM is persistent, data stored in a processor cache is not. Thus, after a failure, only data actually resident in SCM survives. A system using SCM could reduce this restriction with low-level software that flushes data from the processor cache on application or OS failure, or sudden loss of power [136]. While recent work has shown that battery backing is not necessary to tolerate power loss as flush can be completed successfully using a small residual energy window provided by the system power supply [136], acting on failure is still susceptible to data loss caused by either hardware faults or catastrophic failures that prohibit the system from taking post-failure actions. Therefore, unless otherwise stated,

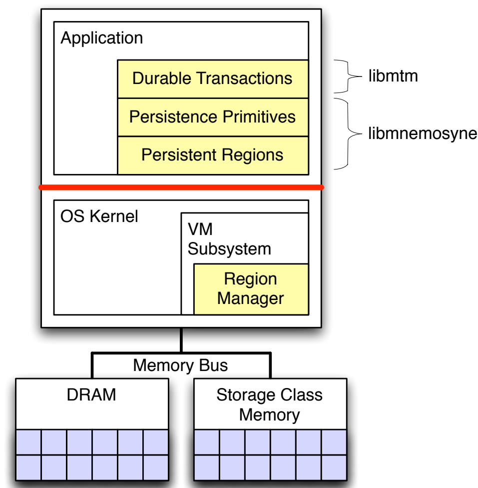


Figure 3.1: Mnemosyne architecture. User-mode components layer on top of a kernel region manager.

we assume that on a system failure, in-flight memory operations may fail, and that atomic updates either complete or do not modify memory.

3.1.2 Persistent Regions

Mnemosyne exposes storage-class memory directly to application programmers through the *persistent region* abstraction: a segment of data that user-mode code can read or write, and that survives application or system crashes. Persistent regions are mapped at constant virtual addresses to support pointer-based data structures. This abstraction makes persistence explicit: only a portion of a process address space persists across restarts. In addition, programs must take explicit steps to guarantee persistence, such as writing data with special instructions or within a transaction to ensure data makes it all the way to SCM. However, data in a persistent region can be read and cached with regular instructions.

Mnemosyne supports both static and dynamic creation of persistent regions. A programmer can declare a C/C++ variable `pstatic`, which tells the linker to place it in a persistent region. Similar to static variables, persistent static variables are initialized once when the program first runs, and then retain their value across invocations. Static persistent variables are useful as their name makes them identifiable in a program at compile time, and they can

serve as named pointers into dynamically allocated persistent regions. A programmer can explicitly create a dynamic persistent region with the `pmap` function, which provides functionality similar to `mmap`. Dynamic persistent regions offer programmers a generic way to store data of any size and structure. Mnemosyne also provides a persistent heap (Section 3.1.3), which enables dynamically allocated persistent variables.

Mnemosyne virtualizes regions by (i) recording the virtual–physical mapping of persistent regions in SCM, and (ii) swapping SCM pages to *backing files* that it allocates when creating a region. Thus, multiple applications can time-share or space-share access to SCM. As we discuss in Section 3.1.5, virtualization prevents a memory leak in one program from monopolizing a finite amount of SCM.

Static persistent regions are most appropriate for programs that allow only a single instance of the program to run per user, such as most office productivity applications and some web browsers. For programs that allow multiple simultaneous instances, Mnemosyne provides an environment variable to indicate which backing file contains the data for this instance.

Mnemosyne lets programmers annotate the target of a pointer type as persistent with the `persistent` keyword, which functions similar to the `const` keyword. The annotation has a shallow effect on the target: annotating a target as persistent does not indicate its members as persistent. The annotation also has no control over whether the data is allocated in volatile or persistent memory. It only serves as an indication to the compiler of the persistence type of the target, which can be used to identify potentially dangerous assignments of volatile address to a `persistent` pointer, and vice versa. This ensures that persistent data, which survives restarts, does not refer to volatile data that is lost. However, this cannot detect when the only pointer to persistent data is *stored in* volatile memory; in this case, the persistent data could be leaked if the program crashes.

3.1.3 Consistent Updates

Persistence requires a mechanism for application programmers to update persistent data without risking corruption after a failure. File systems use a variety of techniques to en-

Method	**Ordering** constraints within update		**Data structures**
	Count	Description	
Single variable update	0	None	flag, pointer
Append update	0	None	log, extent
Shadow update	1	Store modifying reference ordered after stores writing data	tree, bitmap
In-place update	N-1	Stores modifying original data ordered after stores making copy	any

Table 3.1: Methods for consistently updating persistent memory and the number of ordering requirements within an update.

sure consistency, such as shadow updates [23, 160, 42], journaling [184], soft updates [125], restartable updates via backpointers [36], and post-reboot checking [126]. We seek to give programmers similar flexibility in implementing consistency.

A *consistent update* is a transformation that takes data from one state that is consistent according to some program defined integrity rules to another state consistent to those rules; an update may contain any number of store and load instructions. For example, in a hash table, a consistent update may be the addition of a new value.

Ensuring consistency

The primary mechanism for ensuring consistency is *ordering writes*, for example to ensure that new data becomes durable before changing a pointer to reference the new data. Ordering writes ensures that writes complete, that is they reach SCM, in the order they appear in the program. Ordering writes does not prevent issuing a write before the previous ones complete but also does not guarantee that writes will ever complete. This guarantee is provided by *completion*, which ensures writes eventually complete and become durable. Consistent updates that order writes without completion can achieve high throughput as they do not have to wait for writes to reach SCM. However, such updates require special processor support that lets programs express ordering constraints [42]. They also complicate the programming model as they do not guarantee durability. For example, making application-internal state externally visible before an update becomes durable may complicate recovery if the application requires a

permanent record of such an action. We therefore opted for consistent updates that guarantee completion. Such updates leverage the low-latency of SCM to simplify the programming model without significantly hurting performance. If completion latency is a concern, then previously proposed mechanisms that remove completion from the critical path, such as split-phase commit [165] and bounded persistence [164], could be employed.

We identify four common mechanisms for consistently updating data in Table 3.1 in increasing order of flexibility. The more specific mechanisms can provide higher performance for certain data structures, while the more general mechanisms support a wider range of usage patterns. Mnemosyne supports all four methods.

Single variable update. The simplest update is atomically writing to a single variable. This is useful for recording when a program has been initialized or for storing statistics such as counters. These updates are totally ordered with respect to each other.

Append updates. An append update is used by logs and writes new data to empty space after the previous update, thus never modifying existing data. The individual stores comprising an append update are unordered, but separate appends must complete in order. After a failure, an incomplete append (there can be only one) is discarded.

Shadow updates. Similar to append updates, a shadow update writes all data to a new location. Once new data is persistent, the program atomically modifies the reference to the old data to refer to the new data. While there are no ordering constraints between the stores writing the new data, the reference can only be modified after the new data has completed writing. Shadow updates work best for tree-like structures where data is reachable through a single pointer, and must allocate new memory for every update. After a failure, a program must find and release unreferenced new data.

In-place updates. An in-place update can modify any data structure, such as a doubly linked list or a B-tree. It ensures consistency with transactions that undo or complete changes after a failure. As a result, the program must make a copy of either the old data for rolling back, or the new data for rolling forward. Stores updating a data structure must be ordered after stores that create the copy. An alternative to transactions would be to build redundancy in

the data structure in the form of backpointers [36] for detecting partial updates after a crash. Unlike the preceding consistency mechanisms, in-place updates can be used to modify any data structure and therefore enable existing volatile structures to be persistent. However, they perform worse than other consistency methods because they copy data for recovery.

Persistence Primitives

Mnemosyne provides a set of *persistence primitives*, which are low-level operations that enable programmers to implement these consistency mechanisms. At the lowest level, Mnemosyne provides hardware primitives for writing data persistently and ordering writes (described next), which is useful for single variable updates. The system also provides a *persistent heap* for allocating small blocks of memory. Allocated memory and the size of allocations persist across program invocations, so memory can be allocated during one invocation and freed during the next. The persistent heap supports shadow updates by providing space for new data. Mnemosyne also provides a *log facility* to support append-only updates. The log provides a simple double-ended buffer, allowing consistent appends to the tail of the log and truncation at the head of the log.

Mapping consistency onto hardware

A key challenge in implementing consistent updates is ensuring that data reaches SCM in the right order. Unlike disks, which are only accessed through software, SCM is attached to the memory bus and is subject to caching by the processor. While the cache can greatly improve read and write performance, it can also reorder updates to SCM: the cache may evict cached data at any time and in any order. Thus, if a program accesses SCM like normal memory, updates may not be consistent. Prior work on SCM addresses this problem with hardware support for programs to express ordering constraints [42]. However, the proposed hardware requires extensive modifications to processor caches, including an additional *epoch* tag per cache line and the ability to globally flush all caches lines in an epoch.

Mnemosyne's consistent updates mechanism exposes explicit commands to write data

persistently and consistently. It relies on three hardware primitives found in most processors [95]: (i) *write-through stores*, which write data directly to memory rather than to the cache, (ii) *fences*, which prevent subsequent writes from completing before preceding writes, and (iii) *flushes*, which writes a cache line out to memory. Durable updates to persistent memory are implemented as write-through stores, ensuring that they will not be delayed by caching. If a program wishes to separate durability from writing data, so it may re-read data before it becomes durable, it can use a regular store and then flush the data later. Mnemosyne guarantees ordering constraints and completion by issuing a fence between ordered updates. A fence stall-waits for previous updates to complete so it ensures that a later update will not become persistent before an earlier update.

If the application can tolerate data loss caused by hardware faults or catastrophic failures, then Mnemosyne can instead rely on low-level software that flushes volatile state held in the processor cache and buffers to SCM on a failure (Section 3.1.1). Under this approach, Mnemosyne's consistent update primitives degenerate into regular stores. This helps reducing the latency to persist data by removing from the critical path the latency cost associated with ordering and flushing updates to SCM.

3.1.4 Durable Memory Transactions

Mnemosyne provides *durable memory transactions* to support in-place updates. Leveraging recent advances in transactional memory, Mnemosyne provides a compiler to convert regular C/C++ code into transactions. A programmer places the `patomic` keyword before a block of code that updates a persistent data structure, and the compiler produces code that passes all memory references to a transaction system.

The transaction system ensures that all modifications are *atomic* and *durable*, so updates from committed transactions will survive restarts and uncommitted transactions roll back, and *isolated*, so no transaction can see intermediate states of other transactions. Unlike the checkpoint mechanism used by some previous persistent memory systems [42, 172], transactions thus allow multiple threads to concurrently update different data structures.

We initially considered a design where transactions provided only atomicity and durability, leaving the programmer responsible to provide isolation through their own synchronization mechanism such as locks in case needed. However, we eventually abandoned that design as it complicated recovery by requiring the programmer write code to properly synchronize accesses to storage when recovering. We were not entirely convinced that the increased complexity in the programming model was justified by any potential gain in performance. Section 5.2 gives more insight into the problem we faced.

3.1.5 Persistent Memory Leaks

Memory leaks of volatile data may cause a program to crash, but can often be repaired by restarting the program. In contrast, leaks of persistent memory may use all the SCM in a system and be fatal to a program. Mnemosyne provides two mechanisms to help prevent leaks in dynamic persistent regions. First, Mnemosyne requires that programs provide a persistent pointer during allocation to receive the memory. This ensures that when a crash happens right after an allocation completes, memory is not lost as a program can follow the persistent pointer upon a restart to locate the allocated memory. Second, Mnemosyne virtualizes persistent memory by swapping it to files. This ensures that a leak in one program only affects that program and does not reduce the availability of persistent memory to other programs. When leaks do occur, a program can recover by allocating a new persistent region and then copying live data from the existing regions into the new region. Static regions cannot have leaks because the amount of memory is fixed at compile time.

In addition, there are a variety of language-level techniques for preventing leaks, including conservative garbage collection [22], and smart pointers that perform reference counting as demonstrated by other persistent stores [118, 172, 191, 41].

3.2 Implementation

This section describes the implementation of Mnemosyne for Linux. The system consists of (i) kernel modifications to expose and virtualize storage-class memory, (ii) libraries to

Class	API	Description
H/W primitives	flush(addr)	Writes back the cache line that contains the linear address *addr*.
	store(addr, val)	Writes value *val*.
	wtstore(addr, val)	Writes value *val* to SCM.
	fence()	Prevents subsequent writes from completing before preceding writes.
Persistent regions	`pstatic` var	Allocates the variable *var* in the static region.
	pmap(addr, len, prot, flags)	Creates a dynamic region.
	punmap(addr, len)	Deletes part or all of a dynamic region.
	type `persistent` * ptr	Declares the target of the pointer *ptr* as persistent.
Persistent heap	pmalloc(sz, ptr)	Sets *ptr* to point to a newly allocated persistent memory chunk of size *sz*.
	pmalloc_set(sz, ptr, val)	Sets *ptr* to point to a newly allocated persistent memory chunk of size *sz*, and initializes all the bytes of the chunk to the specified value *val*.
	pfree(ptr)	Deallocates the persistent memory chunk pointed by *ptr* and then nullifies *ptr*.
Log	log_create(flags, cbf)	Creates a log.
	log_append(rec)	Writes record *rec* by appending it at the end of the log.
	log_flush()	Blocks until all prior writes to the log reach SCM.
	log_truncate()	Drops any records written to the log.
Durable transactions	patomic {...}	Atomically updates persistent state.

Table 3.2: Summary of Mnemosyne's programming interface, including low-level operations implemented with processor instructions and high-level APIs for managing persistent regions.

implement persistent regions, persistence primitives and the transaction system, and (iii) a compiler/linker that supports persistent variables and transactions. We implement our system for x86-64 Linux version 2.6.33.

The total implementation is comprised of about 450 new lines of kernel code and 10,050 lines of user-mode library code, not including comments. The user-mode code is further broken down into 3,500 lines for implementing persistent regions and persistence primitives, 3,100 lines for implementing the persistent heap, and 3,450 lines for implementing the transaction system.

3.2.1 Hardware Platform

Mnemosyne provides four memory primitives for consistently updating persistent data. As described in Section 3.1.3, we rely on two fundamental operations, *write through* and *fence*, to order updates and to guarantee that updates have reached memory. In addition, Mnemosyne provides a *store* operation, to update persistent memory in the cache, and *flush* to force cached data out to SCM. These allow data written to the cache immediately so it can be returned by later loads, and asynchronously flushed to SCM.

Given the long latency of PCM operations, a goal for the primitives is high throughput by batching multiple operations. We rely on the x86 *write-combining buffers*, which are provided for high-speed data streaming [10]. Write combining allows data to be stored temporarily in a hardware buffer and merged with other data writes to the same block before the processor writes them to memory. The block size is usually that of the cache line, and is 64 bytes on our platform. When the processor eventually writes the complete block to memory, it performs a single burst-transfer bus transaction, which utilizes the full memory bandwidth. Without write combining buffers, writes to the same cache line would require separate partial-transfer bus transactions that do not fully utilize bandwidth and sacrifice performance.

The write-through operation issues streaming writes to the write-combining buffers with the `movntq` instruction. Fences use the `mfence` instruction, which delays execution until write-combining (WC) buffers have been flushed. Thus, any memory access after the fence waits until the data has been written stably to SCM. Since streaming writes on x86-64 use a weakly ordered memory consistency model, the programmer has to be careful to use memory fences before reading data written by write-through operations. For regular writes, `store()` just invokes `mov`, and `flush()` issues a `clflush` (flush cache line). These operations are all implemented as macros for performance.

Finally, while existing x86 hardware primitives provide the necessary functionality to push updates out of the processor, the current prototype implementation is still susceptible to a small failure window. Specifically, current hardware primitives guarantee that a data update reaches the memory controller but do not necessarily guarantee that data hits the

SCM cell. Therefore, a failure that happens after the completion of a hardware primitive but before the memory controller performs the update to SCM causes the update to be lost. One approach to addressing this problem without resorting to processor-side changes would be to extend the memory controller to acknowledge the completion of a data update only after the data hits the SCM cell.

3.2.2 Persistent Regions

The Mnemosyne persistent region abstraction is provided by a combination of kernel support, library support, and compiler support for declaring persistent variables. As shown in Table 3.2, Mnemosyne provides both static and dynamic persistent regions. Static regions contain persistent global variables that are declared and initialized by a programmer at compilation time. A programmer declares a global variable as persistent by annotating it using the `pstatic` keyword. This keyword inserts a compile-time annotation with the `__attribute__ ((section("persistent")))`. The static linker coalesces all persistent variables into a single `.persistent` ELF section in the executable.

Programmers create dynamic persistent regions at runtime by calling the `pmap` function, similar to memory mapping files with `mmap`. Mnemosyne automatically maps dynamic regions created by the program on previous invocations into the address space when it initializes. To prevent newly created sections from being lost if the application crashes, the `pmap` function takes as an in/out parameter a persistent variable to receive the region's address. A programmer deletes a persistent region by calling the `punmap` function, which takes the starting address and the length of the region.

Mnemosyne lets programmers annotate pointer targets with the `persistent` keyword. This keyword inserts a compile-time annotation with the `__attribute__ ((address_space (1)))`. This annotation is interpreted by the Sparse semantic parser [2], which designates such annotated pointer targets in address space *1* and all other pointer targets in address space *0*. Sparse essentially treats pointers with identical target types but different address spaces as distinct types, and warns about code that mixes pointers to different address spaces. This

allows Mnemosyne to identify code that accidentally assigns a pointer to persistent data to volatile data instead (and vice versa).

Persistent regions present two challenges: how to ensure that mappings are persistent, and how to virtualize a finite amount of SCM to support use by many applications. For regular volatile memory, pages are swapped to a shared page file or swap partition and mappings are deleted when a program terminates. However, for persistent regions the mappings of virtual addresses to physical SCM pages must survive system restarts. In addition, data swapped out of SCM must also survive system restarts.

Our implementation follows a layered approach. A kernel-mode layer, the *region manager*, exposes SCM to user-mode code as a memory-mapped file. A user-mode layer, *libmnemosyne*, associates each persistent region with a specific file and records the address of each region.

Region manager. The region manager is an extension of the existing Linux virtual memory system. The region manager creates a new zone of memory for SCM, and all allocations for persistent regions come from this zone using the existing Linux page allocator. We add a new flag, `MAP_PERSIST`, to the `mmap` system call implementation to indicate that a file should be mapped to SCM and not DRAM. The region manager records the list of virtual pages currently stored in SCM in the *persistent mapping table*. This table, stored at the base of physical SCM, stores triples of the form $\langle \mathtt{scm}_{\mathtt{fn}}, \mathtt{i}, \mathtt{p}_{\mathtt{off}} \rangle$, which associate an SCM frame's number $\mathtt{scm}_{\mathtt{fn}}$ with a page offset $\mathtt{p}_{\mathtt{off}}$ in the file identified by inode *i*[1].

The region manager reconstructs persistent regions when the OS boots. It scans the persistent mapping table and (i) updates the Linux page descriptor for each mapped SCM page, and (ii) creates a VFS inode for the backing file of every mapping. The manager places SCM frames not in the table on a free list. After boot, the page descriptors and inodes enable the kernel to evict SCM pages to their proper files. When starting a process with persistent regions, all accesses to SCM pages already in memory are treated as soft page faults that update the page table without copying data from the backing file. During process execution,

[1] Uniquely identifying a file requires two extra pieces of information: device and inode generation numbers, which we omit for clarity of discussion.

the region manager may swap persistent memory pages back to the file system if there is memory pressure.

libmnemosyne. The libmnemosyne library creates and records the persistent regions for a process. Mnemosyne allocates all regions in a one terabyte *reserved range* of virtual address space (easily changed to any power-of-two sized region). This allows a quick determination of whether an address refers to persistent data and prevents persistent regions from conflicting with dynamically allocated address space. The library reserves 16KB in the static persistent region to store a *region table* containing the process's persistent regions. If the region table exceeds 16KB, it can overflow to a dynamically allocated page.

To create a dynamic region, the `pmap` function in libmnemosyne creates an empty backing file and invokes `mmap` to map the file into SCM. In the region table, libmnemosyne records the tuple $\langle r_{addr,len}, b, m \rangle$ that associates the region's starting address $addr$ and length len, with the backing file b, and metadata m, which includes protection flags. The region table also serves as an intention log: libmnemosyne writes a flag indicating the successful completion of a `pmap` operation. When an application starts, libmnemosyne recreates previously allocated persistent regions and destroys partially created ones.

For the static regions containing variables labeled `pstatic`, libmnemosyne creates a new backing file the first time the program executes. It populates the new region with the initial values found in the executable. If a program wishes to discard existing contents of a static persistent region and revert to the data in the executable, a program can delete the backing file and restart.

All the region backing files, including the region table's file, are stored by default in the program's current working directory. This location can be changed via the environment configuration variable `MNEMOSYNE_REGION_PATH`, thus allowing multiple concurrent instances of the same program to use separate backing files.

3.2.3 Memory allocation

Mnemosyne provides a persistent heap [14] in libmnemosyne. Table 3.2 lists the programming interface to this heap. Similar to `pmap`, the `pmalloc` call takes a persistent pointer as an argument to ensure that memory is not leaked if the system fails just after an allocation. The `pfree` call takes a pointer to a persistent pointer as an argument to ensure that the persistent pointer does not continue to point to the deallocated chunk of memory if the system fails just after a deallocation.

We base our implementation on two popular volatile memory allocators, Hoard [21] and dlmalloc [114], which we modify to allocate from a persistent region. While Hoard was originally designed for use in multiprocessor environments, we leverage its superblock-based structure to minimize the persistent state required to track allocations. Hoard splits the heap into superblocks, which are fixed-size regions containing an array of fixed-size blocks (different superblocks may have different block sizes). We modify Hoard to store a *persistent bitmap vector* per superblock to track allocated blocks; allocating memory requires only one write to SCM to set a bit in the superblock's vector. We separate bitmap vectors from allocated data to reduce the risk of corruption [120]. Hoard's indexes, which speed allocation, are in volatile memory and must be regenerated when a program starts. The allocator guarantees atomicity of its operations by logging the write to the bitmap vector and the destination/source pointer.

Mnemosyne uses the modified Hoard allocator for requests smaller than a superblock (8 KB). If the requested block is larger, Mnemosyne falls back to dlmalloc, which we chose for its scalability to large block sizes. Since we expect dlmalloc to be infrequently used, we have not modified it except to add logging to ensure allocations are atomic.

3.2.4 Logging

Mnemosyne relies on a log to make memory allocations and transactions atomic. This log is exposed to programmers by libmnemosyne for implementing append-only data structures. Table 3.2 lists the log programming interface. A program writes to the log with `log_append`,

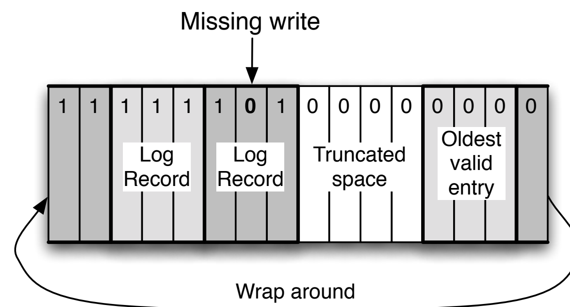


Figure 3.2: Example of missing write detection in the RAWL. Entries written on the current pass through the log have a torn bit of '1', while entries written the previous pass have a '0'. A missing write is detected by an incorrect torn bit.

which writes data but does not guarantee persistence. The `log_flush` call ensures all prior log writes are persistent. Finally, logs are truncated with `log_truncate`. A program can synchronously truncate the log by interspersing truncates with appends in a single thread. Alternatively, the application can asynchronously truncate the log from another thread, which moves the latency of truncating off the critical path. In addition, a program may truncate logs at startup. Mnemosyne's durable transaction mechanism supports all three uses through a runtime flag.

We implemented a high-performance *raw word log* (RAWL) and associated manager that logs uninterpreted word-size values. Such a semantic-free log can be the basis for other semantic-rich logs, such as journals. The log is implemented as a fixed size single-consumer/single-producer Lamport circular buffer, which allows simultaneous appends and truncations without locking [111]. The log manager implements functions to access the log and recover log contents after a failure.

The main challenge in implementing a log is maximizing performance while ensuring that appends are atomic. As logs are always written sequentially, Mnemosyne uses the x86 streaming write instruction `movntq` to write log data. To achieve high performance, these instructions do not guarantee that writes are executed in program order. If the system crashes, later writes may have completed while earlier ones did not. When restarting, the log manager must identify whether all the writes that comprise an append operation completed. This problem is similar to the torn-write problem in databases and file systems, where only part

Algorithm 3.1 Bit manipulation routines for tornbit log

```
procedure ENCODE(log, val)
    tornmask ← 1 ≪ 63
    encoded ← ~tornmask & (log.remainder | (val ≪ log.remainder_nbits))
    encoded ← encoded | log.tornbit
    log.remainder_nbits ← (log.remainder_nbits + 1) & (64 − 1)
    log.remainder ← val ≫ (64 − log.remainder_nbits1) & (64 − 1)
    return encoded
end procedure
procedure DECODE(log)
    tornmask ← 1 ≪ 63
    decoded ← (~tornmask & log.data[index]) ≫ log.remainder_nbits
    encoded_next ← (~tornmask & log.data[index+1]) ≪ (63−log.remainder_nbits)
    log.remainder_nbits ← (log.remainder_nbits + 1) & (64 − 1)
    index ← index + 1
    return decoded_next | decoded
end procedure
```

of a multi-sector write completes. The common solution in file systems is to use two fences: write the data, wait for the data writes to complete with a fence, then write a commit record, and wait for the commit record to complete with a fence. A second option is to append a checksum of the data [148]. However, both these techniques have high overhead. Commit records require two long-latency fences, and checksumming adds substantial overhead to every write.

Mnemosyne instead implements a novel *tornbit RAWL* that requires only one fence. The tornbit RAWL reserves a single *torn bit* in every 64-bit word. This bit has the same value for all writes in one pass over the log buffer, and reverses sense when the log is overwritten. Thus, completely written log entries will have the same torn bit, while incomplete entries will mix values. Figure 3.2 demonstrates the use of torn bits. When Mnemosyne creates a log, it initializes the memory to zeroes. The log manager treats the incoming 64-bit words to be written to the log as a stream of bits. It forms and writes out to the log 64-bit words that are composed of 63 bits taken from the head of the stream and the proper torn bit.

Algorithm 3.1 and shows the bit manipulation routines: `encode` transforms a 64-bit word into a 63-bit word plus a tornbit, and `decode` forms a 64-bit word out of the log contents. On a

`log_flush` the log manager writes out to the log any bits left in the stream. Upon recovery, Mnemosyne locates the head of the log and scans forward until it reaches the end of the log, where the torn bit reverses, or until it finds a log word with an out-of-sequence torn-bit, indicating a partial write.

3.2.5 Limitations

Shared Memory. The recovery mechanisms used for logging and memory allocation execute when a process starts. Furthermore, the allocator caches index information in volatile memory for performance. Therefore, these mechanisms may be unsafe if a persistent region is shared between processes. However, sharing is safe if the processes cooperate to ensure that (i) within each region, only one process writes to a log or allocates from a heap, and (ii) both processes have started and completed recovery before accessing shared data. Thus, producer-consumer style communication, where a single process is responsible for creating and later deleting work items, can be implemented safely.

Wearout. Our prototype implementation does not address wearout of storage-class memory. However, virtualization enables remapping heavily used virtual pages to spread writes to different physical PCM frames. Additionally, our system's structures, such as mapping tables and RAWL, may be relocated to spread writes around. For example, RAWL's tornbits may periodically be shifted to avoid writing 0's and 1's continuously to the same bits. Finally, our system could benefit from work on wear-leveling mechanisms that reduce wearout of PCM when that technology is used as DRAM replacement in main-memory systems [96, 150].

3.3 Durable Memory Transactions

Mnemosyne supports in-place updates with *durable memory transactions*. Updates within a transaction execute to completion, at which point the changes persist across failures, or the changes are rolled back upon restart.

Memory transactions are implemented by three components: a compiler to create transactions from ordinary C or C++ code, a transaction system to store data necessary for recovery, and a transaction log. The *libmtm* library provides the transaction system on top of the persistence primitives in libmnemosyne.

Compiler. We use Intel's STM Compiler [137] to automatically generate object code with calls into a transaction system. The compiler emits compiled code that invokes the transaction system when a transaction begins, commits, and on every memory reference within the transaction.

Transaction system. Mnemosyne transactions are based on Tiny-STM [62] a lightweight software transactional memory (STM) implementation. The Mnemosyne transaction system implements atomicity and durability with write-ahead redo logging, and isolation with encounter-time locking.

With write-ahead redo logging, new values written during the transaction and their addresses are added to a log and also buffered in volatile memory. The transaction system performs a quick range check against the reserved persistent address range and logs only writes to persistent memory. At commit, the log is flushed to SCM, and the new values can optionally be written back. During a transaction, memory at a variable's address still contains unmodified values. When called to read data, the transaction system checks whether the data was modified. If so, it bypasses the new data from the buffer and if not returns the value from memory.

We implement write-ahead redo logging because it reduces the ordering constraints on writes to SCM: the only requirement is that the log is written completely before any data values are updated. In contrast, undo logging, where old values are written to a log and new values are written to memory, would require ordering a log write before every memory update. Ordering is necessary because recovery after a crash must be able to restore old values of all memory locations written by an uncommitted transaction. However, buffering writes through redo-logging has a performance cost: transaction commits are slower because the new values have to be written out to memory, and reads within transactions have to check

if new values exist.

For encounter-time locking, we use a global array of volatile locks, with each lock covering a portion of the address space. A hash function maps addresses to locks in the array. When accessing a memory location, the transaction first identifies the lock that covers the memory address, and if it does not own the lock it tries to acquire it. If successful, the transaction brings the lock in its lock-set and continues with the access. Otherwise, the transaction aborts by releasing all locks acquired, discarding any buffered updates, and writing to the log a mark to indicate the transaction as aborted. When a program starts, Mnemosyne replays all completed transactions by writing the data at the logged address. Locks do not need to persist or be reacquired upon restart as completed transactions are written to the log with locks held and therefore they are properly ordered.

Transaction log. We implement the redo log in persistent memory using a RAWL. The transaction log can be truncated after data values are forced to SCM. We implement both *synchronous* and *asynchronous* truncation. Synchronous truncation forces new values to memory during transaction commit. After flushing the log, the transaction system walks the buffer of modified data and calls `flush` on every address written. It then truncates the entire log. Under heavy load, this prevents the transaction log from growing too large. Asynchronous truncation retains the log after transaction commit, so the latency of committing is shorter. A separate log manager thread consumes the log and forces values out to memory before truncating the log. However, if the log manager thread is unable to execute, program threads may stall until there is free log space.

For better multiprocessor scalability, Mnemosyne keeps a per-thread log. This slightly complicates recovery as Mnemosyne must ensure that transactions are redone in the order executed by the program. Mnemosyne relies on TinySTM's existing global timestamp counter, which is incremented at every transaction completion. Mnemosyne captures a total order over transactions by storing this global counter along with each transaction in the log. During recovery, transactions from different threads are replayed in counter order.

Discussion. The cost of durable transactions is two writes to SCM for every update: once

```
pstatic htRoot = NULL;

main()
{
  if (!htRoot)
    pmalloc_set(N*sizeof(bucket), &htRoot, 0);
}

hash_update(key, value)
{
  patomic {
    pmalloc(&bucket, sizeof(*bucket));
    bucket->key = key;
    bucket->value = value;
    insert(bucket);
  }
}
```

Figure 3.3: Example use of a durable memory transaction.

to store recovery information in the log, and once to write the data itself. Other consistent-update mechanisms may perform better. But, they come at the cost of increased complexity, such as recovery code to replay logs for append-only updates, garbage collection for memory lost during shadow updates, and explicit fences to order updates. For example, to atomically modify the payload value stored in a tree node, a programmer may use a shadow update that swaps the old node with a new node that contains the new value and points to the same subtree as the old node. Using a shadow update saves the extra log writes that would be otherwise needed by a durable transaction to log the old value.

3.4 Programming Examples

In this section we present several code examples that exemplify Mnemosyne's programming interface (Table 3.2). The examples work at different levels of the interface: ranging from high-level transactional updates (Figure 3.3) down to low-level persistence primitives (Figure 3.4) We defer discussion of how we used Mnemosyne to persist data in real-world applications to Section 3.5.

Figure 3.3 demonstrates how to construct a persistent hash table and use the `patomic` block language construct to form a transaction that updates the table. To construct the table, we first declare the root of the hash table as persistent through the `pstatic` keyword to provide a persistent name that locates the hash table upon a restart. When the program starts for

```
paxos_logprop(prop, value)
{
  log_append(PROPOSAL);
  log_append(prop);
  log_append(value);
  log_flush();
}
```

(a) Log update

```
bst_insert_right(parentnode, key, value)
{
  pmalloc(&parentnode.right,
          sizeof(parentnode.right));
  wtstore(&parentnode.right.key, key );
  wtstore(&parentnode.right.value, value);
  wtstore(&parentnode.right.left_valid, 0);
  wtstore(&parentnode.right.left, NULL);
  wtstore(&parentnode.right.right_valid, 0);
  wtstore(&parentnode.right.right, NULL);
  fence();
  wtstore(&parentnode.right_valid, 1);
}

bst_recover(bst)
{
  foreach node in bst
  {
    if (node.right && !node.right_valid)
      pfree(&node.right)
    }
    if (node.left && !node.left_valid)
      pfree(&node.left)
    }
  }
}
```

(b) Shadow update

Figure 3.4: Example use of low-level persistence primitives.

the first time, we atomically allocate space for the hash table, initialize to zero, and assign it to the persistent root through the `pmalloc_set` function. Recall that `pmalloc_set` takes a persistent pointer as an argument to ensure that the allocated memory is accessible even if the system fails just after allocation. To update the hash table, we use a memory transaction that atomically allocates space for the new bucket, sets the key-value pair of the bucket and inserts the bucket into the hash table. The transaction guarantees atomicity so that a failure mid-flight the transaction does not leave the hash table in an inconsistent state like missing a bucket. In summary, upon a restart, the hash table is accessible via the persistent named root, and contains all key-value pairs successfully written to it.

Figure 3.4a shows an example use of the persistent log primitives to log a proposal during the *accept request* phase of the Paxos protocol [112]. We first append to the log a header that identifies the type of the append sequence, then append the values forming the sequence, and finally issue a flush to ensure all prior log writes are persistent. Upon recovery the log returns only appends that were completed successfully, and we rely on the headers to identify the boundaries of each sequence.

Figure 3.4b shows an example use of hardware primitives to insert a new child node into a binary tree using a shadow update. We first atomically allocate space for the new node and attach it to the parent. Then we update the key and value of the node using write-through stores, initialize the rest of the node's fields, issue a fence to ensure that the writes are performed to SCM, and finally update the `right_valid` flag of the parent node to indicate the right child node contains a consistent key-value pair. To prevent space leakage, upon recovery `bst_recover` traverses the tree and frees any node that is linked to a tree node but does not represent a valid node (i.e., the node's corresponding valid flag stored in its parent node is false).

3.5 Evaluation

The goal of Mnemosyne is to abstract storage-class memory at low latency for program use. We evaluate three questions about the system:

1. *Does it work?* Can existing programs benefit from persistent memory, and does Mnemosyne provide persistence across crashes?

2. *How fast is it?* How does Mnemosyne perform, both in latency and throughput, across a range of workloads, and how beneficial are Mnemosyne's mechanisms?

3. *With which technologies can it work?* How does Mnemosyne's performance depend on the latency of the underlying SCM technology?

While programmers may program directly with persistence primitives, these primitives require a sophisticated understanding of recovery protocols similar to databases or file systems [23, 131]. Durable transactions provide a much simpler interface to persistent memory and require few application changes, so we concentrate our evaluation on transaction performance.

3.5.1 Methodology

As we do not have access to a form of storage class memory that can be plugged into a commodity system and access from user mode, we emulate SCM using DRAM by adding delays to model SCM's performance. We base our model on the performance characteristics of phase-change memory (PCM) , which is the nearest to commercial availability. Because there is a wide variety of projections for PCM's performance, and the specific design of the memory system can have a great impact on performance [115], we limit our model to the most important aspect of performance: slow writes.

To account for PCM's slower writes relative to DRAM, we introduce a delay after each write into the hardware access macros. The emulator adds delays on operations that already go to DRAM (not the cache), so the delay is only for the *additional latency* of PCM. For cacheable writes (`store` operations) we insert the delay on the subsequent `flush`, while for non-cacheable ones we insert the delay in-place. We also insert the delay after each memory fence to account waiting for outstanding writes to PCM. For sequential write-through (`wtstore`) we model write bandwidth by inserting a proper delay after the write sequence completes to limit the effective bandwidth.

In all cases we implement the delay with a loop that reads the processor's timestamp counter (TSC) in each iteration. The loop continues until the requested delay has elapsed. In calibration tests, we found that inserted delays are at least equal to the target delay, and that our bandwidth model is accurate to within 4%.

Our emulator does not account for additional latency on loads. However, the primary benefit of SCM is fast, durable updates, and our workloads focus on write rather than read performance. Furthermore, many loads will hit the cache, so only misses will incur a penalty from PCM. Also, our model does not account for the effect of cache evictions or read-after-write bank conflicts, where reads may be queued behind long writes to the same bank, thus increasing read latency.

To compare Mnemosyne against other uses of PCM, we constructed an emulator, *PCM-disk*, for a PCM-based block device. Base on Linux's RAM disk (`brd` device driver), PCM

disk introduces delays when writing a block. We model block writes using sequential write-through operations as described above, and mount an ext2 file system.

We performed our experiments on an Intel Core 2 quad-core based machine equipped with 4GB of physical DRAM (accessible via the DDR2-800 interface) running our modified x86-64 Linux version 2.6.33 kernel at 2.5GHz. All tests add 150 ns of extra latency and are limited to 4GB/s of write bandwidth unless otherwise noted. We estimated write bandwidth based on projections provided by Numonyx [53]. For all our experiments we report averages of at least five runs.

3.5.2 Applications use of Persistent Memory

Applications that can immediately benefit from persistent memory are difficult to find: the long latency of writing to disk has taught programmers to avoid frequently committing data. We expect that a common use will be for applications to create a useful in-memory data structure, such as a tree or hash table, and then make it persistent by allocating it from a persistent heap and wrapping updates in transactions. For reliability and to support conversion between program versions, the program should periodically export serialized version of the structure.

We model this use of persistent memory by adapting programs that *already* maintain a fast in-memory data structure for frequent access and a separate version for consistent, durable updates. We converted two programs that take different approaches to persistence for use with Mnemosyne: OpenLDAP and Tokyo Cabinet. One program frequently commits data to disk using a storage manager, and the other periodically snapshots a whole data structure to disk.

OpenLDAP. OpenLDAP is an implementation of the Lightweight Directory Access Protocol (LDAP). OpenLDAP supports a number of storage backends; the default is *back-bdb*, which provides transactional storage using Berkeley DB. An alternative, *back-ldbm*, also uses Berkeley DB but without transactions; instead, it periodically asks Berkeley DB to flush dirty data to disk to minimize the window of vulnerability. To improve query performance, each backend

maintains its own cache of data outside Berkeley DB [37]. We believe that such read-mostly caches can benefit from lightweight persistent memory: the backing store can be removed, leaving only a persistent cache.

To test this hypothesis, we modified the back-ldbm backend to remove Berkeley DB and to make the cache persistent with durable transactions. The cache is organized using an AVL tree, which we make persistent by allocating nodes with `pmalloc` and placing `atomic` blocks around updates in four places.

While we do not generally encourage keeping pointers to volatile memory in a persistent region, we found this useful in OpenLDAP. The cache entries of the original back-ldbm and our back-mnemosyne store pointers to a description of each attribute, which is kept by the front end in volatile memory. Instead of modifying the frontend to keep those descriptions in persistent memory, we found it more convenient to keep the descriptions in volatile memory and have the persistent cache entry keep pointers to those volatile descriptions. Because the volatile descriptions become stale after a restart, we augmented each cache entry with a version number that is used to determine whether the persistent pointer is up-to-date.

TokyoCabinet. Tokyo Cabinet is a high-performance key-value store [89]. It stores data in a B+ tree and periodically calls `msync` on a memory-mapped file to flush modified pages to disk. These syncs can reduce performance by 96 percent [196], so they are rarely invoked. As a result, the application loses unsaved data after a crash.

We modified Tokyo Cabinet to allocate its B+ tree in a persistent region and perform updates in durable transactions. While we re-used the existing update code, in several cases we had to duplicate functions to create separate persistent and volatile versions with different allocators. We completely removed the persistence code that calls `msync`. We also removed the locks used for synchronizing concurrent accesses to the tree and relied on transactions for concurrency control.

Memcached. Memcached is a distributed in-memory key-value store. It is mostly used by dynamic web applications as an in-memory object cache between the application and the database backends. As a cache, it helps improve application performance by speeding

Application	Backend	Workload	Updates/s
OpenLDAP	back-bdb on PCM-disk	SLAMD	5428
	back-ldbm on PCM-disk		6024
	back-mnemosyne		7350
Tokyo Cabinet	`msync` on PCM-disk	64B	19,382
		1024B	2,044
	mnemosyne	64B	42,057
		1024B	30,361
Memcached	memcached	64B	86,900
		1024B	81,370
	memcachedb on PCM-disk	64B	21,760
		1024B	11,340
	memcached-mnemosyne	64B	35,260
		1024B	30,320

Table 3.3: Update throughput for OpenLDAP, Tokyo Cabinet, and Memcached. Key size for Tokyo Cabinet and Memcached is 16B.

up access to frequently used objects and alleviating the load on the database backends [1]. However, as a volatile cache, it has to be refilled and warmed up after a restart. For large-scale applications that depend heavily on caching, cache warmup may take hours, which can have a negative impact on the availability of the application [61].

We therefore modified Memcached to make it persistent. Memcached stores data in a hash table. When the table becomes full, subsequent inserts evict existing objects in a least-recently used (LRU) order. Our modifications were quite straightforward: we modified Memcached to allocate the hash table in persistent memory through `pmalloc`, wrapped all seventeen places accessing the table within durable memory transactions, and replaced all locks used for synchronizing concurrent access to the table with transactions.

We compare our modified persistent Memcached against the original Memcached but also against MemcacheDB, which is a distributed key-value storage system that is compatible with the memcached protocol. MemcacheDB uses Berkeley DB as a storing backend.

Performance. Table 3.3 lists throughput of the OpenLDAP server for a workload that adds new records. We used the SLAMD distributed load generation engine to exercise three versions of the server: (1) *back-bdb*, the default unmodified transactional backend with the cache and Berkeley DB, (2) *back-ldbm*, the unmodified back-ldbm backend with the cache and

Berkeley DB, and (3) *back-mnemosyne*, our modified backend based on back-ldbm. We set the cache sizes large enough to avoid evictions due to capacity. We used a LDIF template to generate a workload of 100,000 directory entries. OpenLDAP is configured to run with 16 threads (4 threads per core) as advised by its tuning manual.

While the three backends perform similarly, back-ldbm offers a lower level of reliability than the other two backends. This close performance arises because PCM is fast enough that the time to write updates is a small fraction of the total time to service a request. These results demonstrate that with Mnemosyne it may not be necessary to create specialized structures optimized for persistence, such as Berkeley DB's tables. Instead, standard in-memory data structures (in this case an AVL tree) provide simpler programming, in addition to durability and consistency, all with similar or better performance than a highly tuned storage engine.

Table 3.3 also shows the throughput of 64-byte and 1024-byte insert/delete queries with Tokyo Cabinet for single-thread configurations. As a comparison, we ran the standard implementation of Tokyo Cabinet on our PCM-disk emulator and configured it to save data with `msync` after every update. Mnemosyne was about 2 – 15 times faster in these tests, while at the same time providing *stronger* consistency guarantees than the `msync` version, which can suffer from torn writes if the system fails while flushing pages. For four-thread runs, we found that the throughput of Tokyo Cabinet Mnemosyne degrades by 9% because of increased contention on the tree, causing transactions to abort. The throughput of Tokyo Cabinet on PCM-disk increases by 10%, but is still far below Mnemosyne.

Finally, Table 3.3 also shows the throughput of 64-byte and 1024-byte insert/delete queries. We used the *memslap* load generation engine to exercise three versions of Memcached: (1) *memcached*, the unmodified volatile Memcached, (2) *memcachedb*, a persistent distributed key-value store that is compatible with the memcached protocol, and (3) *memcached-mnemosyne*, our modified persistent Memcached. To saturate Memcached, we ran both memslap and Memcached on the same machine and configured memslap to run with four threads with each thread making 128 connections.

Mnemosyne was about 2.5 – 2.7 times slower than the volatile Memcached, while *mem-*

cachedb was about 4 – 7 times slower. For all three versions, throughput decreases as the object size increases. This is due to larger transmission packets for larger objects, which results in larger memory copies from the kernel into memcached's hash-table buckets.

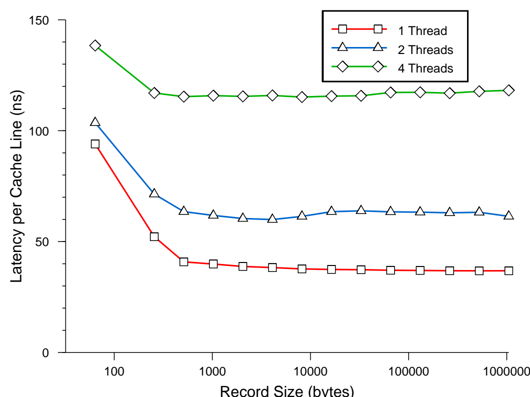
The two persistent versions also pay a persistence cost that is proportional to object size. In the case of Mnemosyne, the time spent in the transactional runtime increases from 69% for 64-byte objects to 77% for 1024-byte objects. This is due to the higher number of calls into the runtime to instrument updates and the longer time to commit larger objects. In the case of *memcachedb*, the drop in performance is primarily due to increase in the time spent in Berkeley DB and the file system, which increases from 66% for 64-byte objects to 78% for 1024-byte objects. Overall, Mnemosyne can help bridge the performance gap between a persistent and a volatile version of Memcached. Still though, the performance hit of persistence may be prohibitive for a production deployment of a persistent Memcached. However, it should be noted that this is performance slowdown for a write-intensive workload (100% writes). Evaluating Memcached on a more representative read-mostly workload (90% reads), we found that Mnemosyne is about 1.3 times slower than the volatile Memcached.

Reliability. We validate that Mnemosyne works correctly by injecting synthetic failures. We intentionally crashed OpenLDAP in the middle of a transaction, and verified that after every restart, the data affected by the transaction were still available. In addition, we wrote a crash stress program, which uses transactions to perform random updates to memory using a known seed. We verified that after a crash, memory contains the correct random values. Finally, we tested the torn-bit feature of the RAWL by injecting bit flips into the log before a crash. In all cases, Mnemosyne correctly recovered after the crash.

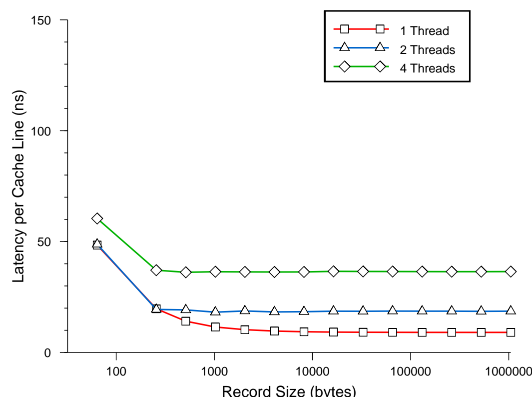
3.5.3 Microbenchmark Performance

Hardware Primitives

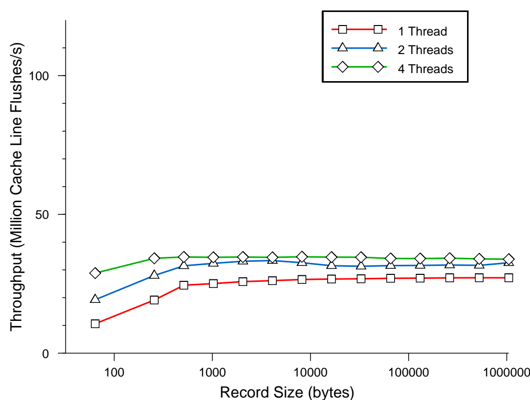
Figure 3.5 shows the latency and throughput to write and flush a cache line for different number of threads and record sizes using either regular writes or streaming writes. We report latency and throughput for the native system without adding any extra latency to emulate



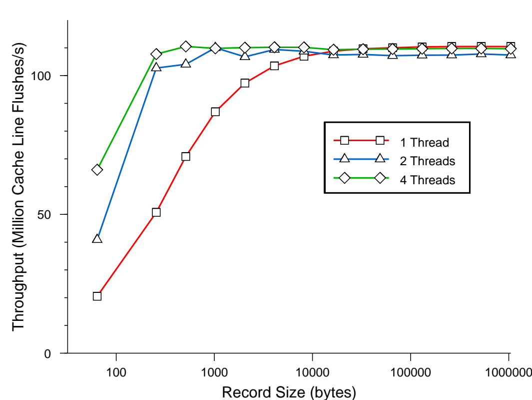
(a) Regular-write latency



(b) Streaming-write latency



(c) Regular-write throughput



(d) Streaming-write throughput

Figure 3.5: Latency and throughput to flush a cache line.

writes to PCM. We took the measurements using a microbenchmark that allocates 64 MB of memory for each thread and subdivides it into equally sized cache-aligned records. Record sizes ranged from 64 bytes to 64 MB. For regular writes, each thread writes to every cache line in a record using x86 regular stores (`movq`), issues a memory fence (`mfence`) to order subsequent flushes after the stores, flushes the entire record by performing a `clflush` on each cache line of the record followed by a memory fence (`mfence`) that waits for the flushes to complete, and then repeats the process with the next record. For streaming writes, each thread writes to every cache line in a record using x86 non-temporal stores (`movntq`) followed by a memory fence (`mfence`) that waits for the stores to complete, and then repeats the process with the next

record.

As we see, the latency for flushing both regular and streaming writes increases with the number of threads due to contention on the memory bus. Similar behavior is seen in throughput, which scales sublinearly with the number of threads, eventually reaching a plateau due to saturation of the memory bus. Then, looking at how latency and throughput scale with record size, we observe that latency per cache line flush decreases as the record size increases and flattens out after record sizes of 512 bytes. This suggests that flushing writes above 512 bytes can be more efficient. Throughput follows a similar trend. Finally, latency and throughput for flushing streaming writes is between 47 - 57% lower than latency for regular writes. This difference reflects a trade-off between performance and programmability as streaming writes use a weakly ordered memory consistency model, which requires extra effort from the programmer to properly order reads (Section 3.2.1).

Persistence Primitives

Programmers wishing to persist in-memory data structures today have a wide variety of choices. We evaluate two common approaches: (i) use a storage manager for data storage and caching, and (ii) serialize the data structure to a file. With a storage manager such as Berkeley DB [46], a program can commit small changes to a data structure, such as adding a record to a hash table. We compare the performance of a simple hash table [39] using Mnemosyne transactions for persistence against using Berkeley DB's hash table. We store the database files on our PCM-disk emulator. In both cases, data is committed to storage on every update.

Figures 3.6 and 3.7 compare write latency and update throughput of Mnemosyne to Berkeley DB for different number of threads and record sizes. Deletes are introduced at the same rate as writes to ensure steady progress. Update throughput is aggregate throughput of writes and deletes. For single threaded runs and transactions that update records smaller than 2048B, Mnemosyne's direct access of memory achieves a write latency that is almost six times better than Berkeley DB. Mnemosyne's performance for small transactions is limited by the cost of (i) transaction instrumentation in the code, which adds a function call to every load

and store, (ii) fences to force the log to memory and (iii) flushes to force data to memory. With a microbenchmark, we measured the cost of instrumenting and logging each word written as 190 ns when the transaction's write set size is smaller than 128 cache lines. For larger write sets, this cost increases linearly with the number of distinct cache lines written. The cost of committing a transaction, which consists of a fence and flushing data out to SCM, adds up to 250 ns per distinct cache line flushed. These costs represent the overhead of supporting in-place updates. A hash table insert of 64 bytes requires on average 15 updates to 5 distinct cache lines, for a total cost of 4.3 μs. With larger data sizes, Berkeley DB's optimizations for disk-like performance, such as large sequential writes and infrequent fences (once per block in PCM-disk), give it better write latency.

With multiple threads, Mnemosyne achieves 10-14x improvement in update throughput compared to Berkeley DB. Mnemosyne improves throughput almost linearly with the number of threads, without hurting write latency. The slight increase in write latency is due to contention on the global timestamp counter in the transaction system, which is relatively more expensive for short transactions. In contrast, Berkeley DB does not scale beyond 2 threads. We found this is due to contention on the centralized log buffer, which becomes the serialization bottleneck as I/O latency becomes shorter. Also, Berkeley DB's throughput improvement with 2 threads comes at the cost of increasing write latency, possibly due to group commit, which is not necessary with Mnemosyne's fine-grain memory transactions. Finally, while Berkeley DB's write latency is lower than Mnemosyne's for values larger than 2048 bytes, the Mnemosyne's throughput is higher because it is the aggregate of writes and deletes, and delete latency remains almost constant for Mnemosyne as value size increases.

An alternative approach, often used for less structured data, is to serialize the data into a buffer and write it to a file. For example, productivity applications including word processors use this approach for periodic *fast saves*. We compare the cost of maintaining a red-black tree with 128 byte nodes in persistent memory against the cost of keeping it in DRAM and periodically serializing it and storing it in a file. Using small data sizes maximizes the overhead of transactions, because there is not much data to write out between fences.

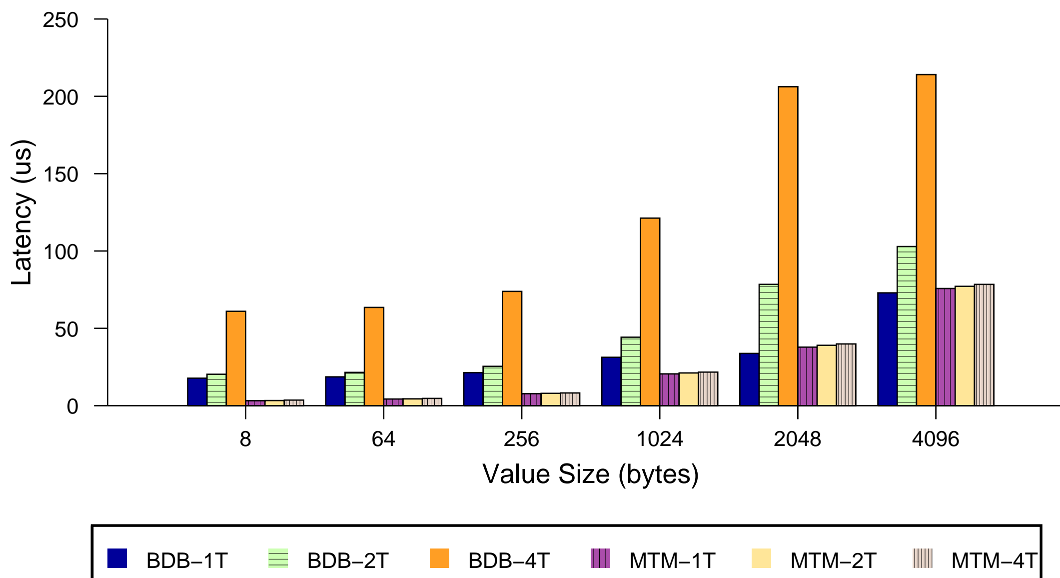


Figure 3.6: Write latency for a hashtable with durable transactions compared to Berkeley DB.

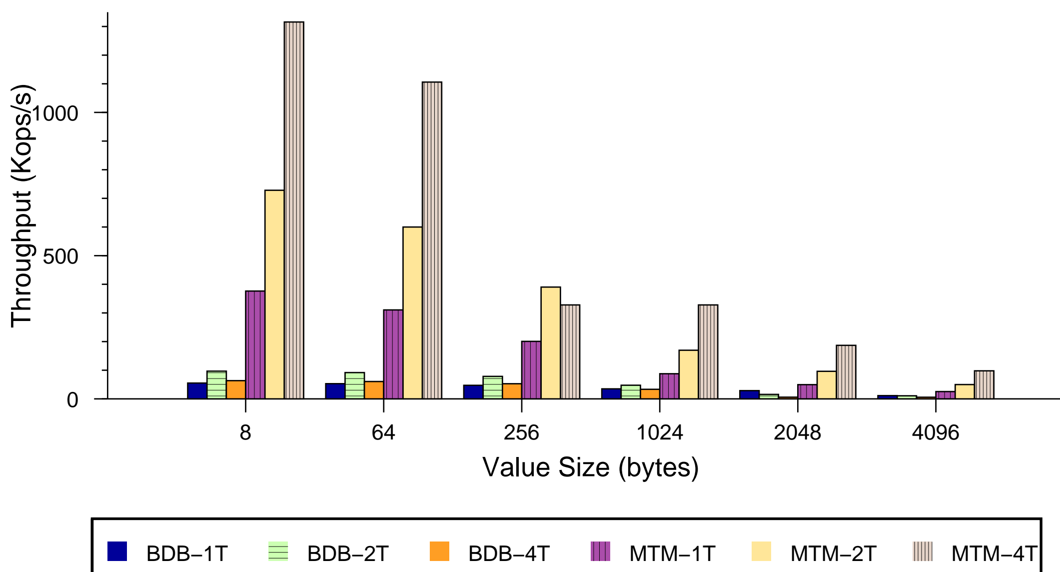


Figure 3.7: Update throughput for a hashtable with durable transactions compared to Berkeley DB. Numbers above the bars are the absolute updates per second for the Berkeley DB single-threaded version.

Tree Size	Insert Latency	Serialize Latency	Inserts per Serialization
1 K	4.7 µs	517 µs	189
8 K	5.1 µs	3,413 µs	1,345
64 K	5.5 µs	33,859 µs	9,202
256 K	5.8 µs	143,776 µs	24,788

Table 3.4: Performance of Mnemosyne updates and Boost serialization of red-black trees. The right-most column shows the number of Mnemosyne updates that can be performed in the time for a single Boost serialization of the tree.

Record Size (B)	**8**	**64**	**256**	**1024**	**2048**	**4096**
Base (MB/s)	17	128	416	881	1088	1244
Tornbit (MB/s)	34	227	591	929	1045	1093

Table 3.5: Throughput of base and tornbit RAWLs.

Table 3.4 compares the cost of updating the tree with Mnemosyne transactions against the cost of periodically serializing it using Boost [24] and storing it on PCM-disk. Tree updates cost 5-6 µs with Mnemosyne, and on average 10 percent of the tree can be updated for the cost of serializing and storing the tree just once. Thus, even data structures with high rates of change can more efficiently be made persistent with transactions than by serializing and storing.

Optimizations

Mnemosyne implements two optimizations to reduce the cost of transactions: the torn bit in the RAWL, and asynchronous log truncation. Table 3.5 compares performance of torn bit against an implementation of the RAWL that writes a commit record, with a separate fence. For log records smaller than 2048 bytes, the torn-bit log performs up to 100 percent better. Above 2048 bytes, the torn bit log performs *worse* than a separate commit record. The cost of the fence is fixed, while the cost of bit manipulation to implement torn bits scales with the amount of data. Thus, for large records, the cost of manipulating bits to make space for the torn bit is larger than the cost of a single extra fence. As Mnemosyne targets smaller transactions, the torn bit log is a valuable optimization but should be omitted if large transactions are expected.

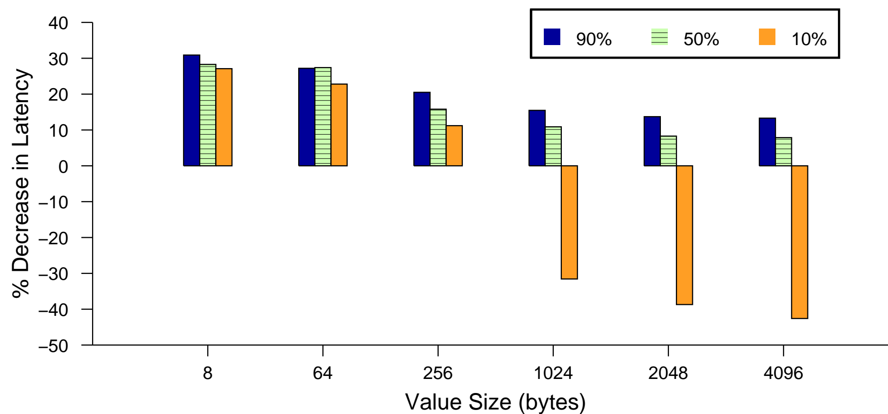


Figure 3.8: Decrease in write latency of hashtable with asynchronous over synchronous truncation.

Asynchronous log truncation may reduce latency and improve performance under low load because it moves flushing modified data off the critical path. Instead, a separate thread processes the log to flush all modified data to PCM, after which it truncates the log. Figure 3.8 compares performance when the hash table thread is idle 90, 50, and 10 percent of the time. We find that for 50 and 90 percent idle time, the truncation thread can keep up with the active thread and achieves a reduction of 7-31% in write latency. However, with only 10 percent idle time, the active thread stalls for extended periods while the truncation thread flushes data, which can increase latency by up to 42% for 4KB writes. Thus, this optimization is most helpful for transactions with a moderate duty cycle.

Reincarnation Cost

There are two reincarnation costs associated with Mnemosyne: (i) the cost to reconstruct persistent regions when the OS boots, and (ii) the cost to remap persistent regions to a process' address space, scavenge the persistent heap, and replay all completed but not flushed transactions when a program starts. For (i), we measured the worst case scenario cost, which is to reconstruct a persistent region for each SCM frame. This takes approximately 734 ms for 1GB of SCM, indicating that reincarnation contributes less than 1s overhead to the OS boot process. For (ii), we found the overall cost in our workloads to be less than 100 ms. Specifically,

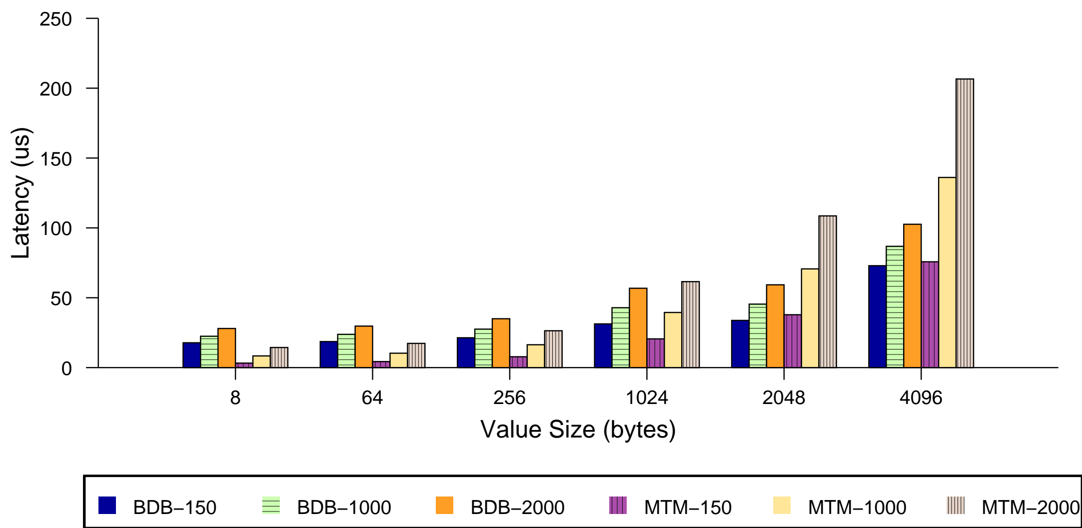


Figure 3.9: Mnemosyne's performance relative to Berkeley DB for different memory access latencies.

we measured the cost to remap the persistent regions to be about 1.1 ms, while the cost to scavenge the persistent heap region and reconstruct the heap's volatile indexes was about 89 ms. This high cost is due to the incremental allocation of memory needed by the volatile indexes, which results to a large number of `brk` system calls. We believe this cost could be reduced via bulk allocation or lazy construction of the indexes but have not implemented any of these optimizations. Finally, we found the cost to replay a completed but not flushed transaction ranges between 3 to 76 μs. In the case of synchronous truncation, the number of completed but not flushed transactions is bounded by the number of threads, which for 4 threads results in a worst case cost of about 300 μs.

3.5.4 Sensitivity to Memory Performance

Mnemosyne exposes persistent memory directly to programs as memory. If SCM latencies are long, then applications may perform better treating SCM like disk, with associated optimizations to overlap I/O and computation and to optimize for sequential access. We evaluate the impact of different memory latencies by comparing the performance of Mnemosyne

transactions on a hash table to Berkeley DB, which optimizes for disk-like performance.

Figure 3.9 shows the relative performance of Mnemosyne over Berkeley DB as a function of data element size for three different latencies: 150 ns, 1000 ns, and 2000 ns. For small data sizes, Mnemosyne is always faster because it needs to write much less data. However, for longer latencies the benefit is much lower: 200 percent better performance for 1000 ns latencies and 100 percent for 2000 ns latencies. Furthermore, the benefit drops off faster with larger data sizes: at 2000 ns, Berkeley DB and Mnemosyne perform the same for 1024 byte inserts.

Thus, Mnemosyne is most useful when SCM latencies are close to those of DRAM, because the benefit of modifying small amounts of data outweighs the latency of access. For larger latencies, SCM may best be treated as a disk and accessed through the file system. Should someone prefer using Mnemosyne though, then our optimizations that remove latency off the critical path become even more evident. For example, under low load (10 percent idle) and 2000 ns latency to SCM, asynchronous truncation would be able to reduce write latency for 1024 and 2048 bytes by 70% and 60% respectively, enabling Mnemosyne to still perform better than the disk-based interface.

3.6 Summary

Programmers trained in the era of disks have learned that frequently updating persistent state should either be avoided or be handled by a database engine. Storage class memory presents new opportunities for fast data persistence that obsolete these rules. Mnemosyne provides lightweight persistent memory, and common in-memory data structures can be made persistent using durable transactions. Thus, programmers can create a single data structure optimized for memory, rather than designing separate in-memory and update-optimized persistent representations. Compared to past work on persistent object systems, Mnemosyne provides greater flexibility by not requiring C++ objects, and lower latency by persisting data directly to memory at the granularity of a a single update rather than a whole page.

4 AERIE: APPLICATION-SPECIFIC FILE SYSTEMS

We propose distributing the storage stack to create a flexible, high-performance storage architecture by mapping SCM into client processes. This provides two key benefits: (i) low-latency access to data by removing layers of code, and (ii) flexibility by enabling applications to define their own file-system interface and implementation without extending the kernel. With direct access, a program reading a file can locate the file contents and read the data directly, without calling the kernel. In addition, an application with fixed-size files can pre-allocate storage in contiguous extents to lower the cost of locating file contents. Prior efforts at user-mode file systems, such as FUSE [175], provide flexibility at a heavy performance cost from context switching and copying data.

Based on this idea, we designed the Aerie architecture to expose file-system data stored in SCM directly to user-mode programs. Applications link to a file-system library that provides local access to data and communicates with a service for coordination. The OS kernel provides only coarse-grained allocation and protection, and most functionality is distributed to client programs. For read-only workloads, applications can access data and metadata through memory with calls to the file-system service only for coarse-grained synchronization. When writing data, applications can modify data directly but must contact the file-system service to update metadata.

We implement two file-system interfaces on the same layout with Aerie: a POSIX-style system and one optimized for small-file access through a put/get interface. Through experiments, we show that the POSIX-style system performs much better than existing user-mode file systems and between only 15% slower than a kernel file system without consistency guarantees and 35% faster on average than ext3. The specialized key-value file-system interface performs up to 86% faster than the fast kernel file system. Thus, distributing file system functionality to client processes allows flexible implementations that dramatically improve performance.

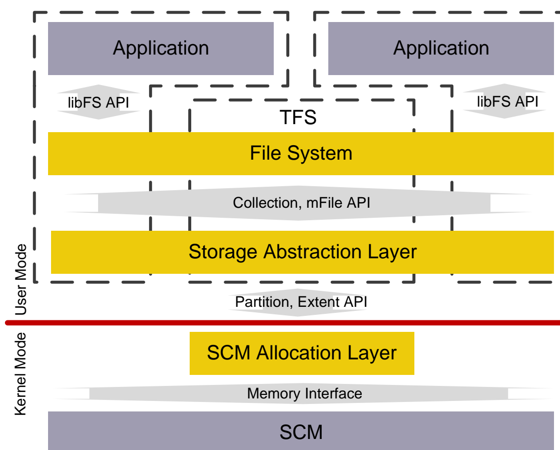


Figure 4.1: Aerie architecture. Functionality is distributed between application processes, a trusted service, and the kernel.

4.1 Design

Aerie is a local storage service that enables programs to store and share data through a user-mode file system. Our goals for Aerie are:

1. *Flexibility* for applications to customize the file system interface, policies, and layout.

2. *High performance* to reap the benefits of low-latency storage memory.

3. *Protected sharing* between processes to enable flexible application structuring and composition.

The main enabling mechanism for our goals is direct access through memory to file-system data and metadata from user mode. Although direct access enables our first two goals, it is at odds with our third goal of protected sharing and raises several other challenges. Protected sharing requires mediating each file system access to enforce file system permissions. For performance, we use hardware memory protection when possible, which poses the challenge of translating file system permissions to memory protection.

Designing a file system for flexibility and direct access raises many challenges, of which we focus on four. First, how should file systems be structured to provide maximum flexibility to applications while still providing access to shared data? Existing file systems have a

single implementation per system, but direct access enables each program to have a different implementation. Second, if clients have direct access to data, how can we ensure a malicious or buggy application does not corrupt file system structures? Third, how do we provide concurrency control between applications efficiently? Kernel file systems use locks in shared memory and provide only limited locking capabilities to user-mode code. Finally, how can we minimize communication for coordination between clients to keep performance high?

4.1.1 Assumptions

We designed Aerie based on assumptions about the features of future SCM products and application behavior. First, we assume that hardware provides user-mode code with low-latency, protected access to SCM. This can be achieved by placing SCM directly on the memory bus, which allows access through normal load/store instructions protected by virtual memory hardware [42, 41, 136]. Alternatively, protected user-mode DMA that restricts the set of memory and SCM addresses can also be used [30].

Second, our failure model assumes that any state stored in volatile memory such as the hardware cache or main memory may be lost but any state written to SCM persists. We therefore depend on hardware to provide an atomic update of at least 64 bits [42, 144]. Similar to past systems [41, 188], we also rely on the device to implement necessary reliability mechanisms to address wear and that such mechanisms are robust to malicious attacks that aim to overwhelm the anti-wearout mechanism [150]. Without such hardware support, the system must be limited to clients that are trusted not to inflict excessive wear.

Third, we assume that most sharing is sequential rather than concurrent. Programs with concurrent access may be better served by a centralized file system. When concurrent sharing occurs, Aerie provides correct behavior but with reduced performance.

Finally, we assume programs may exhibit malicious or buggy behavior such as stray writes. We therefore do not trust the client library to correctly modify file-system metadata. However, we assume that trusted code in the kernel or service is as reliable as existing file system code (which can also corrupt data in memory [198]).

4.1.2 Architecture

Aerie distributes file system functionality for direct and protected access to storage memory. Figure 4.1 shows the Aerie organization. The SCM manager in the kernel allocates SCM to file systems and constructs page tables that map SCM into processes with the necessary protection. Programs link to the libFS library that provides the file system API. The trusted file system (TFS) service provides integrity for metadata updates and concurrency control between processes.

Aerie relies on hardware protection to enforce access control over file system data. This allows the client library to service most file-system operations directly from SCM without contacting the service or the kernel. For example, when an application opens a file, the library accesses directory contents in SCM to locate the file and can then read file data directly from SCM as well.

Thus, Aerie provides two paths to storage: (1) a fast, untrusted code path, and (2) a slow, trusted code path. The untrusted code path executes within any user process and relies on virtual memory hardware to limit access to data. Thus, reading files can be provided completely by the code in libFS. The trusted code path invokes TFS to support any operation on file-system state that cannot be enforced in hardware. For example, enforcing integrity constraints, such as reference counts, cannot be left to clients.

In order to support multiple file systems, a system may have multiple implementations of the library and the service, one for each file system implementation. The kernel code, though, is common to all file systems. In addition, a single file system may have multiple libFS implementations optimized for different workloads.

Abstractions and Interfaces

To encourage flexibility, we architect Aerie in three layers, each providing a specific service. Like ZFS layers (ZPL, DMU, and SPA) [23], these layers allow lower-level services to be reused by multiple higher-level file systems or file-system interfaces. Table 4.1 lists the layers of Aerie and the main operations of the layer.

Abstraction	Function	Operations
SCM Allocation		
Partitions	Contiguous memory that can be mapped to virtual address space	`allocate, free, mount`
Extent	Contiguous memory with same access rights	`create, delete, mprotect`
Storage Abstraction		
mFiles	Mapping of offsets to memory locations	`create, delete, addExtent, removeExtent, protect`
Collections	Grouping of storage objects	`create, delete, add, remove, protect`
Locks	Concurrency control	`request, release, revoke`
File System		
Files, Directories	Standard POSIX file systems	`create, unlink, read, write, ...`

Table 4.1: Aerie layers and their supported abstractions and operations.

Storage-class-memory allocation layer. At the bottom level, the SCM allocation layer does the minimum work required for protection: it records and enforces resource usage. To achieve this, it exposes storage in the form of *memory partitions* and *memory extents*, which are close to hardware and provide primitive operations upon which higher-level interfaces are built.

A memory partition is a contiguous region of virtual addresses mapped to SCM, and is used for coarse-grain allocation of physical SCM to a file system volume. Partitions are named by their starting address, and the allocation layer exposes a list of partitions and their owners similar to a mount table.

A memory extent is a range of memory within a partition associated with protection rights and higher-level software assigns their location and size. Memory extents are similar to standard file-system extents with the addition of protection and are used by higher layers to store data within an object, such as file contents. While each extent is contiguous virtually, higher-level objects constructed out of extents, such as files, may not be as they can combine non-adjacent extents through an indirection structure. Extents are named by a descriptor that encodes their partition and offset within the partition. In addition to methods to create and delete extents and partitions, the SCM allocation layer provides a method to change the protection of an extent with the `mprotect` operation.

The SCM allocation layer provides a simple ACL representation for higher levels to specify

permission on file-system data. These ACLs are stored for each extent.

Storage abstraction layer. This layer adds structure and synchronization to the raw memory exposed by the allocation layer. It provides low-level methods that can be used to implement functionality commonly found in file systems, thus allowing higher-level flexibility.

The storage abstraction layer implements two basic abstractions of memory: the *memory file* (*mFile*), which maps offsets to extents, and the *collection*, which groups objects (mFiles or other collections). Both mFiles and collections provide the same access to all members (extents for mFiles and elements for a collection). These abstractions are the building blocks for files, directories, and other file-system metadata structures.

The storage abstraction layer also provides *distributed concurrency control* through a hierarchical lock service to synchronize access to shared data across processes. The lock service lets clients request a lock for shared or exclusive access to an object. It also offers *hierarchical intent locks* [78, 192], which can make locking more efficient when resources can be organized in a hierarchy such as a directory of files. We describe this more in later sections.

File-system interface layer. The top layer implements a file system API. Similar to Pilot [154], this level provides a name space above lower-level collections and mFiles and can expose familiar interfaces to create, delete, or rename files. While the lower level provides the mechanisms for storing, retrieving, and protecting data, the interface layer provides the policies of how and when to access data. The FS interface layer assigns protection both to storage objects that clients can read directly and to metadata limited to the TFS. The interface can either be a standard POSIX interface or tailored to application needs, such as a tag-based lookup mechanism [169].

Components

While Aerie seeks to provide as much functionality within a client process as possible, some file-system features require a third party. Cooperation between mutually distrustful programs

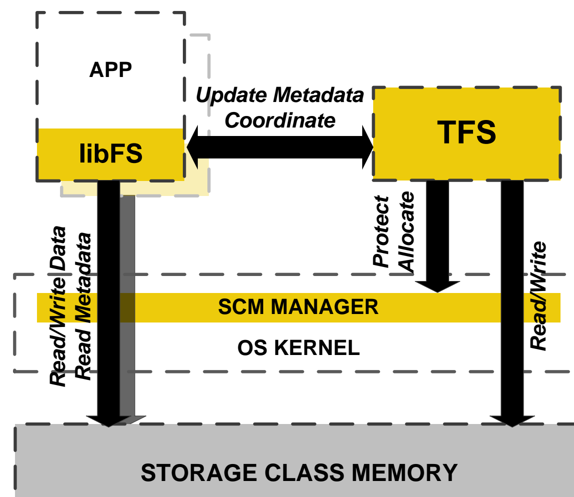


Figure 4.2: Aerie design components (solid colored). Arrows show communication between components.

requires a trusted entity to enforce synchronization and integrity [167]. Thus, Aerie distributes the file system among three components.

libFS Client Library. Applications link against a libFS library for each file-system interface they use. The library provides functionality from both the FS interface and storage-abstraction layers needed to find and access data: lookup to map file names to file metadata, and indexing to translate a file offset into a byte in memory. It also implements logic to invoke a trusted service.

Trusted File System (TFS) Service. Functionality that requires a trusted third party (*i.e.*, integrity and concurrency control) but *not* privileged hardware access execute in the TFS service, which includes code both from the storage abstraction and FS interface layers. The service also provides complete file system functionality on data for which memory protection is too coarse. The service executes as a user-mode process accessed via RPCs.

SCM Manager. Operations requiring hardware privileges, such as modifying memory permissions or virtual address mappings, must execute in the kernel. The SCM manager

provides the storage allocation layer, which contains only low-level mechanisms that are independent of high-level file system organization.

4.1.3 File System Design

In previous sections we described the layered architecture and basic components of Aerie. Here, we discuss how to build a complete file system with this architecture.

Naming. Aerie implements a namespace by mapping each directory to a collection. The collection maps a name to the identifier of an mFile (for a file) or a collection (for a subdirectory). While namespace operations are metadata heavy, operations that read metadata can be serviced directly and safely by the library to avoid communication with the TFS service on every file access. Thus, directory operations such as traverse, read, and open are handled by locating collections in memory and reading them directly.

Protection. We use hardware page-level protection to implement file-system permissions. We precompute the permissions for each piece of file data and then construct a page table for a process based on the permissions it receives. In contrast, a normal file system checks an ACL on every open, whereas we compute the effective ACL for the user and put those permissions in the page table.

Although hardware-enforced protection enables us to efficiently enforce permissions, it may be too rigid to directly represent the whole spectrum of file-system permissions. For example, while read permission on a file is equivalent to read permission on a page containing the file data, this does not hold true for directory contents: *list* and *traverse* both require reading the directory contents, but only *list* lets the client see all the names. Thus, clients with traverse only should not be able to access the page containing directory contents.

A file-system implementation can therefore choose which memory to protect with hardware and which to protect in software. Data with simple read/write permissions corresponding to memory protection can be made available to libFS through memory. If not, then libFS must call into the TFS service to access the object.

Integrity. In order to prevent corruptions by untrusted code, only the TFS service applies modifications to file system objects. Clients are only trusted to modify file contents but not file system objects, such as mFiles and collections. Instead, they create a log of their operations, similar to a file system journal, and ship the journal to the TFS. The TFS then validates the entries in the journal are legal, protected by locks, and preserve invariants, such as link counts and free/allocated status. Finally, the TFS applies them as a transaction by forcing the log to SCM and then updating data structures.

Similarly to a kernel-mode file system though, the service is vulnerable to corrupting itself. The file system and storage abstractions can use existing techniques such as checksums, replication, sealing pages through memory protection, or periodic checks of metadata.

Concurrency. Aerie performs concurrency control over storage objects using the distributed hierarchical lock service. File systems use hierarchical locks to grant a client access to a subtree of the name space. A client can read all the files within a subtree without communicating with the server, which improves performance. A client requests a lock that covers an object, and can use that until it is done or the lock is revoked. The TFS requires that clients hold a lock covering an object when sending a metadata update to the TFS. Clients, though, can treat locks as advisory, because the data is already available through memory. In addition, a client may change the granularity of locks internally, such as by assigning byte-range locks to different threads while the process holds a lock on the entire file.

The file system library is free to provide applications any degree of consistency and concurrency semantics. For example, Windows file systems provide mandatory locking, while POSIX file systems do not, and GFS provides atomic append operations [69]. A client library can choose when to acquire locks and how long to hold them to implement different consistency levels [78].

Clients can batch updates together while holding a lock. This amortizes the cost of communicating with the server. Internally, a client should structure operations as short-lived because a lock may be revoked on short notice. When a lock is revoked, the client must ship to the service any buffered metadata updates covered by the lock before releasing the lock.

This is required because TFS does not accept metadata updates that are not covered by a lock in order to preserve invariants during concurrent updates. If a client does not revoke the lock when requested, TFS forcefully releases the lock, which clients treat similarly to media removal.

Durability. The TFS guarantees atomicity and durability of metadata operations, but the libFS clients must ensure durability of file data by forcing it out to SCM. To provide atomicity when an operation modifies multiple metadata objects, the TFS provides atomicity using write-ahead redo logging. For example, writing new data to a file may require the file system to allocate and insert multiple extents into the file's storage object, all of which go to a persistent journal first.

Failures. Aerie must handle four types of failure. First, media failure may happen when SCM hardware is physically damaged. Such failures can be addressed using existing techniques such as checksums and replication. Second, client failures may happen when a client crashes due to a software error. Aerie addresses client failures by revoking locks held by the client and discarding any outstanding metadata updates a failed client has not yet shipped to the service. This guarantees metadata invariants but allows client data to be lost. Third, the whole system may fail due to a software or hardware error, or power failure. Upon system restart, the service restarts and can perform recovery from the log. Third, the service may fail. We treat this similar to a system failure, and require that the TFS and all clients restart.

4.1.4 Summary

Aerie provides flexibility through simple abstractions that support a variety of file system interfaces. It is structured in three layers, which are distributed through the kernel, client libraries, and a trusted file-system service. It prevents a faulty client from corrupting data by centralizing metadata updates at the TFS, which ensures clients can only corrupt file contents (similar to current systems). Aerie uses a distributed hierarchical lock service to synchronize

Subsystem	# Lines	Subsystem	# Lines
Infrastructure services	3800	Storage objects	10220
SCM manager	650	PXFS	2660
Distributed Lock Service	2910	KVFS	340
Total	20580		

Table 4.2: Aerie implementation size.

access to files, which allows many operations to be performed without communication and many updates to be batched together.

An alternative design to direct access to SCM is to cache an entire file system in DRAM and use SCM only to log updates. We see two challenges to this approach: (1) recovery may be slow, as it requires rereading data from another medium (flash or disk) and re-applying a long log, and (2) SCM device scalability may be better than DRAM and lower power so it may not be economically justified to keep the whole file system in DRAM [115].

The Aerie design shares strong similarities with many distributed file systems that allow clients to access a storage service directly. However, it relies on hardware for access, which only provides read and write operation, rather than software, which can provide a much richer interface [135, 71]. The trust model also differs from past systems relying on a shared block device, because clients cannot be trusted to make correct changes to metadata [182].

4.2 Implementation

We implemented an Aerie prototype on Linux 3.2.2 for x86-64 processors. In this section we present the implementation of the two lower-level layers of Aerie, the SCM allocation layer and storage abstraction layer. We defer the discussion of two file system interfaces to Section 4.3. Table 4.2 summarizes the amount of code comprising Aerie's major components. Kernel-mode code is written in C, while user-mode code is written in C++.

4.2.1 Infrastructure Services

Aerie relies on low-level mechanisms for inter-process communication and for consistently updating data in SCM.

Interprocess Communication. We use remote-procedure call (RPC) implemented using sockets on loopback interface for communication between clients and the server . The server is multithreaded and can handle multiple RPC requests concurrently. Batching of metadata operations at a client (Section 4.2.4) helps take RPC off the critical path for most operations. A lightweight RPC implementation that leverages shared memory and hardware synchronization could further help reduce the cost of communication [18].

Persistence Primitives. We borrow the persistence primitives from Mnemosyne [188] to support consistently updating file system structures in SCM in the presence of failures. We use regular x86 instructions and provide three basic operations: (1) *wlflush* uses x86 `clflush` to write and flush a cache line out of the processor cache into SCM for persistence, (2) *bflush* uses x86 `mfence` to flush the processor write-combining (WC) buffers into SCM for persistence, and (3) *fence* uses x86 `mfence` to order writes to SCM.

We use these primitives to implement our higher-level consistency mechanisms and a persistent log for redo logging. Writes to the log are done using x86 streaming instructions (which buffer writes in WC buffers and enable high bandwidth for sequential writes). Flush of the log writes to SCM is done through bflush.

4.2.2 SCM Manager

The SCM manager is a kernel component that provides the storage-class memory allocation layer. Its responsibility is allocation, mapping, and protection of SCM.

Allocation. The SCM manager is designed for allocating a small number of large static memory partitions. It allocates contiguous regions of physical memory using first-fit. The SCM manager stores a table listing each partition and an access control list indicating who can

modify or access the partition, typically the TFS. As with all data structures in Aerie, the SCM manager stores the partition table in SCM and uses persistence primitives to assure consistent updates. The SCM manager delegates extent allocation to user mode so that user-mode file systems can implement the allocation policy they prefer. User-mode code allocates an extent by choosing a location within a partition and then calling the SCM manager to record the location and protection information of the newly allocated extent.

Mapping. Once allocated, a partition can be mapped into any process with the `scm_mount_partition` API. In order to reduce the overhead of page tables, the SCM manager uses a linear mapping of physical addresses that can be computed from a single virtual base address, and maps SCM at the same virtual address in all processes. Thus, mounting a partition does not actually map it into an address space but instead ensures that ensuing page faults will lazily create the page table. This effectively treats the page table as a giant software TLB, similar to Mach's pmap structure [153]. As a result, page tables are dynamic structures that need not be stored in SCM.

The SCM manager further reduces the space overhead of the page table by aggressively sharing page tables between processes. All processes with the same access to files—those with the same user and group IDs—share the entire page table.

Protection. The unit of protection in Aerie is the extent. We store extents in a radix tree corresponding to the page-table layout. Each extent consists of a starting address, length, and a 32-bit ACL identifier. The 30 higher bits represent a group identifier (GID) and the lowest 2 bits represent the memory protection rights (read, write). The `scm_create_extent` API takes a starting address and length in pages and creates an extent structure. The `scm_mprotect_extent` changes the protection on an extent. Only processes with write access to a partition can manipulate extents. At run time each process inherits and maintains the user's group memberships in a hash table. On a page fault, the manager uses the GID of the extent as a key in the hash table to quickly decide if the process has access to the extent.

Changing permissions on an extent is more expensive than changing permission on a file

because permissions must be changed for all clients of the file system. To avoid synchronously modifying many page tables, the SCM manager instead invalidates portions of the page table mapping the affected extents (if they were valid), and allows them to be faulted back in later. Thus, clients implicitly communicate with the kernel to reload mappings when protection changes.

We borrow a technique from single address space operating systems to handle page faults [33]. When a page fault occurs, the SCM manager computes a new page table entry from the linear mapping and the permissions stored in the extent tree. On a processor architecture with support for separating protection from addressing [107], only a single page table would be needed.

4.2.3 Distributed Lock Service

The storage abstraction layer contains two services: distributed concurrency control and storage objects. We discuss concurrency first and storage objects in the next section.

We implement distributed concurrency control with a centralized lock service executing in the TFS service. The lock service provides multiple-reader, single-writer locks identified by a 64-bit identifier. Our implementation derives from prior lock services for storage systems [182, 97, 192, 73, 78, 121]. However, because our lock service is intended for a single machine, we do not replicate the service for fault tolerance. Aerie does not use Linux's futexes [57] because it must be able to revoke locks.

Clients access the lock service via a local clerk. When a client thread requests a lock, the clerk invokes the lock service to acquire a global lock that synchronizes the client with other processes. The clerk then issues a local lightweight mutex that client threads use to synchronize within the process. When another process requests conflicting access to the lock, the service calls the clerk back to revoke the lock.

The clerk may hold the lock after a thread releases the local mutex. It releases the global lock when it has not been used recently or when the lock service calls back to revoke the lock. If the lock is in use when a callback arrives, the clerk prevents additional threads from

acquiring the local mutex and releases the global lock when the local mutex is released. Clients of the lock service are responsible for preventing deadlocks by ordering or preempting locks.

Hierarchical locking. Aerie assigns a unique global lock to every object, such as a collection or mFile. Like a futex, the ID of an object can be used as the name of a lock [57]. The lock manager provides three modes for each lock: *explicit*, meaning the lock covers only a single object; *hierarchical*, meaning it covers the object and its descendants, and *intent*, meaning that the object is not locked, but a descendant may be. When acquiring a hierarchical lock clients can access members of a collection without additional locks.

The clerk in libFS implements the hierarchical locking logic. If it holds a hierarchical lock, the clerk answers requests for locks on descendant objects locally and issues local mutexes. For example, a client can lock a directory of files using a global lock and then acquire local mutexes on individual files. The clerk de-escalates in response to revocations [97]. When another thread requests conflicting access to a resource protected by a hierarchical lock, the clerk will request locks lower in the hierarchy and release the high-level lock.

Protection. An unresponsive client can deny service to the file system due to bugs or malicious behavior. This occurs in any system with mandatory file locks, such as Windows. Aerie addresses denial-of-service by attaching a *lease* to each lock that must be renewed by the clerk [73]. Leases bound the time processes wait behind a lock for an unresponsive client that does not renew its lease implicitly releases the lock and allows other processes to proceed. Furthermore, Aerie can limit the number of locks a process may hold to reduce contention [28].

4.2.4 Storage Objects

The storage abstraction layer provides objects upon which file systems can build. The implementation is shared between the TFS server, which is responsible for writing to objects, and libFS, which provides read access to objects and write access to object contents.

Data Structures

We provide simple implementations of mFile and collections that are sufficient to show the use of the abstractions. We use C++ template polymorphism to decouple the implementation of the interface from the implementation of layout and consistent updates. Thus, other implementations are possible with the same interface and could provide better performance or less space overhead.

Each storage object is identified by a 64-bit integer (a *storage object ID*). The six least-significant bits encode the type of the object and the remaining 58 bits encode the virtual memory address where the object is stored. This encoding enforces a minimum object size of 64-bytes and provides 64 different types. As a result, accessing an object requires no lookup of its address, but it cannot be relocated in memory. We did not find the lack of relocation to be an issue in the file systems we implemented. Objects can grow arbitrarily large without having to be relocated because they are not linear regions of virtual memory but a structure composed of multiple extents. Storage objects expose a buffer to store metadata from the file system interface layer. They are allocated from extents with other objects sharing the same protection.

Collections. The collection object provides an associative interface for storing key-value pairs. We implement collections as a linear hash table that is packed into extents. The hash table stores key-value pairs in which a key is an array of bytes of arbitrary length and the value field stores a 64-bit storage object ID. When the hash table fills, we attach additional extents and rehash some existing elements into the new extents. We perform consistent updates using shadow updates, so new extents are allocated and populated and then linked into the hash table with a single 64-bit atomic write to a pointer. We delete items by marking them using a tombstone key. When the number of tombstones rise above a configurable threshold, we rehash the live key-value pairs into a new table and then update the collection's header to point to the new table with a single 64-bit atomic write.

mFile. The mFile object provides access to a range of bytes starting at a specified offset. We implement the mFiles as radix tree of indirect blocks that point to fixed-size extents. Larger extents are broken into pieces when added to the tree.

Protection

The file-system layer assigns protection to a storage object and the storage layer must propagate that protection to the memory containing the file objects using the `scm_mprotect_extent` API.

However, memory and file systems do not have perfectly compatible protection models: memory typically grants read or read/write access, while files may have write-only access. In addition, metadata may have semantically richer permissions, such as directory list and search. The file system interface maps each granted permission to the protection it enables and each denied permission to the protection it disables. The granted permissions remaining after removing denied permissions are then mapped to memory. This process ensures that permissions requiring conflicting protection are properly enforced. For example, granting write-only access to a file allows the write permission, which is enabled by read/write protection, but disallows the read permission, which prevents read-only and read/write protection. Thus, the file data would be set to no-access protection.

The untrusted library can directly access any storage memory allowed by protection. Since protection is stricter than permissions, the library calls into the TFS service for any operations allowed by file system level permissions but prevented by memory protection, as in the case of write-only files.

Concurrency

The storage layer associates each storage object with a global lock. Clients acquire the lock in read mode when they read objects directly, and in write mode for metadata updates performed by the TFS. The storage service checks with the lock service to verify that clients hold the appropriate lock on the object before it performs any updates.

Hierarchical locking overview. Aerie file systems may use hierarchical locks by organizing storage objects in a tree hierarchy through collections. When a client acquires an hierarchical lock on a collection, it implicitly locks any other storage object accessible through the collection.

When a storage object is member of multiple collections, such as a file hard linked to multiple directories, standard hierarchical locking no longer works. This happens because there are multiple paths leading to the object, but an hierarchical lock locks a single path. The classic solution would be to lock each collection from which the file is accessible [78]. However, this approach requires finding those collections, which introduces complex bookkeeping. Instead, we follow a novel locking protocol where clients do not need to lock each collection but instead explicitly lock just the object. We next discuss our hierarchical-locking protocol in more detail and explain how it locks objects that may belong to multiple collections.

Protocol details. Each object keeps a membership count that states what type of lock should be used to lock the object: if the membership count equals one then an implicit hierarchical lock may be used, otherwise an explicit lock must be used. For the membership count we maintain the following invariant:

Invariant 4.1. *An object's membership count never underestimates the number of collections the object is member of.*

Thus, an object's count may overestimate the actual number of collections. This is safe because in such a case a client will conservatively deduce that it must acquire an explicit lock on the object. The service updates the membership count when it adds or removes an object from a collection. To maintain the above invariant, the membership count is increased before adding an object into a collection, and it is decreased as the last step when removing an object from a collection.

When a client wants to lock an object, it first obtains intent locks on all the ancestor objects of the object to enable conflict detection with future hierarchical locks by other clients. Then the client uses the object's membership count to deduce the type of lock to acquire. If the

object's membership count equals one then it tries to acquire an implicit hierarchical lock on the object, otherwise it tries to acquire an explicit lock. Recall from Section 4.2.3 that an hierarchical lock may be acquired implicitly if the client already holds a globally visible hierarchical lock on an ancestor of the object.

When TFS updates an object per client's request, it first verifies that the client holds the right lock on the object. If the object's membership count equals one then TFS accepts both implicit locking through an hierarchical lock and explicit locking. When using an implicit lock to perform an update, TFS requires the client prove that it has an hierarchical lock that covers the object by providing the list of collections between the locked and mutated object. If the object's membership count is greater than one then TFS accepts only explicit locking.

Proof sketch: We informally show the correctness of our locking protocol by considering the two possible race conditions:

1. *Insertion race:* A client inserts an object into a collection, and $\mathit{MembershipCount} = 1$ transitions to $\mathit{MembershipCount} > 1$. The client either holds an (implicit) hierarchical lock or an explict lock on the object. A client that concurrently accesses the object may see $\mathit{MembershipCount} = 1$ and try to acquire an hierarchical lock. The client serializes because the object is still only accessible through a single collection, which is already locked by the client inserting the object into the new collection. If the client reaches the object through another collection then by our membership invariant we know that $\mathit{MembershipCount} > 1$ so the client will try to acquire an explicit lock.

2. *Removal race:* A client removes an object from a collection, and $\mathit{MembershipCount} > 1$ transitions to $\mathit{MembershipCount} = 1$. The client holds an explicit lock on the object because $\mathit{MembershipCount} > 1$. A client that concurrently accesses the object may see $\mathit{MembershipCount} > 1$ and try to acquire an explicit lock. The client serializes because the client removing the object already holds the explicit lock. The client holding the explicit lock releases the lock only after it removes the object and sets $\mathit{MembershipCount} = 1$. When a concurrent client sees $\mathit{MembershipCount} = 1$, the

Abstraction	Operation	Log-record fields	Invariants
Extent	`createExtent`	fsid, eid, slot	4.3
	`deleteExtent`	fsid, eid, slot	4.3
mFile	`createMFile`	fsid, mfid, slot	4.4
	`deleteMFile`	fsid, mfid, slot	4.4
	`addExtent`	mfid, boff, eid	4.2, 4.3
	`removeExtent`	mfid, boff, eid	4.2, 4.3, 4.5
Collection	`createCollection`	fsid, cid, slot	4.4
	`deleteCollection`	fsid, cid, slot	4.4
	`addObject`	cid, key, oid	4.2, 4.4, 4.6, 4.7
	`removeObject`	cid, key, oid	4.2, 4.4, 4.5

Table 4.3: Log records of storage-object operations and their type-specific fields. Refer to Table 4.4 for invariants.

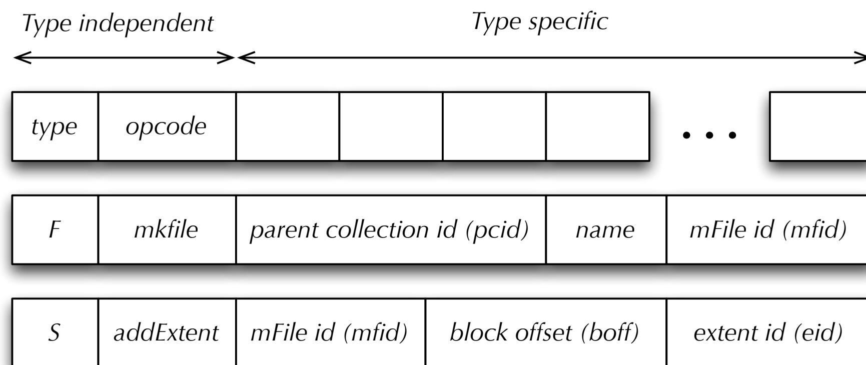


Figure 4.3: Log record format.

. . .					
F	mkfile	0x0AAA...06 (parent id)	"foo" (name)	0x0AFA...08 (mfile id)	
S	createMFile	0x0BAE...0A (freeset id)	0x0AFA...08 (mFile id)	413 (slot hint)	
S	addObject	0x0AAA...06 (collection id)	"foo" (key)	0x0AFA...08 (mFile id)	
. . .					

Figure 4.4: Log fragment corresponding to a file create for *foo*.

client can safely deduce that there is a single collection leading to the object and can therefore try to acquire the hierarchical lock. □

Invariant 4.2. *Any object modified is covered by a lock:* $$\mathtt{modifyObject}(\mathtt{obj}) \Rightarrow \mathtt{hasLock}(\mathtt{obj})$$
Invariant 4.3. *An mFile object maps only allocated extents:* $$\mathtt{createExtent}(_, \mathtt{eid}, _) \Leftrightarrow \mathtt{addExtent}(_, _, \mathtt{eid})$$ $$\mathtt{deleteExtent}(_, \mathtt{eid}, _) \Leftrightarrow \mathtt{removeExtent}(_, _, \mathtt{eid})$$
Invariant 4.4. *A collection object maps only allocated mfile or collection objects:* $\mathtt{createMFile}(_, \mathtt{mfid}, _) \Rightarrow \mathtt{addObject}(_, _, \mathtt{mfid})$ $\mathtt{createCollection}(_, \mathtt{cid}, _) \Rightarrow \mathtt{addObject}(_, _, \mathtt{cid})$ $\mathtt{addObject}(_, _, \mathtt{mfid}) \wedge \mathtt{type}(\mathtt{mfid}) = \mathtt{mFile} \Rightarrow \mathtt{createMFile}(_, \mathtt{mfid}, _) \vee \mathtt{exists}(\mathtt{mfid})$ $\mathtt{addObject}(_, _, \mathtt{cid}) \wedge \mathtt{type}(\mathtt{cid}) = \mathrm{Collection} \Rightarrow \mathtt{createCollection}(_, \mathtt{cid}, _) \vee \mathtt{exists}(\mathtt{cid})$ $\mathtt{deleteMFile}(_, \mathtt{mfid}, _) \Rightarrow \mathtt{removeObject}(_, _, \mathtt{mfid})$ $\mathtt{deleteCollection}(_, \mathtt{cid}, _) \Rightarrow \mathtt{removeObject}(_, _, \mathtt{cid})$
Invariant 4.5. *No orphan extents or objects left behind:* $\mathtt{removeObject}(_, _, \mathtt{mfid}) \wedge \mathtt{linkCount}(\mathtt{mfid}) = 1 \wedge \mathtt{type}(\mathtt{mfid}) = \mathtt{mFile} \Rightarrow \mathtt{deleteMFile}(\mathtt{mfid})$ $\mathtt{removeObject}(_, _, \mathtt{cid}) \wedge \mathtt{linkCount}(\mathtt{cid}) = 1 \wedge \mathtt{type}(\mathtt{cid}) = \mathrm{Collection} \Rightarrow \mathtt{deleteCollection}(\mathtt{cid})$ $\mathtt{deleteMFile}(_, \mathtt{mfid}, _) \Rightarrow \neg\exists \mathrm{eid} : \mathrm{eid} \in \mathtt{extentSet}(\mathtt{mfid})$ $\mathtt{deleteCollection}(_, \mathtt{cid}, _) \Rightarrow \neg\exists \mathrm{oid} : \mathrm{oid} \in \mathtt{objectSet}(\mathrm{oid})$
Invariant 4.6. *No duplicated keys exist in a collection:* $$\mathtt{addObject}(\mathtt{cid}, \mathtt{key}, _) \Rightarrow \mathtt{key} \notin \mathtt{keySet}(\mathtt{cid})$$
Invariant 4.7. *Collections do not form cycles:* $$\mathtt{addObject}(\mathtt{pcid}, \mathtt{key}, \mathtt{cid}) \wedge \mathtt{type}(\mathtt{cid}) = \mathrm{Collection} \Rightarrow \mathtt{noCycle}(\mathtt{pcid}, \mathtt{cid})$$

Table 4.4: Storage-object invariants. Symbol _ matches any value. Predicate noCycle is defined by the file system using the storage objects.

• • •				
F	write	0x0AFA…08 (mFile id)	526 (block offset)	8192 (nbytes)
S	createExtent	0x0AAE…0A (freeset id)	0x0AFD...00 (extent id)	634 (slot hint)
S	addExtent	0x0AFA…08 (mFile id)	526 (block offset)	0x0AFD...00 (extent id)
S	createExtent	0x0AAE…0A	0x0AFE...00	739
S	addExtent	0x1FAA…08	527	0x0AFE...00
• • •				

Figure 4.5: Log fragment corresponding to a file write of size 8192 bytes at block offset 526. Each extent has size 4KB.

Integrity

File-system interfaces and the storage layer cooperate to guarantee the integrity of metadata. The file-system interface enforces high-level invariants, such as ensuring that rename operations do not cause cycles in the namespace. The storage layer enforces storage-object invariants, such as ensuring that mFiles map only allocated extents.

The TFS server performs all modifications to file system objects in order to prevent clients from violating these invariants. To avoid frequent synchronous calls to the service for every metadata update, clients buffer their updates locally in a log that they send to a server periodically (similar to delayed writes) or when they must release a global lock.

The log is implemented by the storage abstraction layer. It contains file-system interface operations similarly to Coda [102]. A file-system operation is an atomic transformation of file-system objects that maintains file-system invariants. A file-system operation may comprise multiple storage-object operations, which as a whole must maintain storage-object invariants. That is, a log record for a file-system operation may nest multiple log entries of storage-object operations. Figure 4.3 shows the format of a log record. Each record has a type-independent header identifying the operation and a sequence of type-specific fields. The same figure also shows two example log entries: (i) a file-system log record `mkfile` that creates a file *mfid* named *name* in directory *pcid*, and (ii) a storage-object log record `addExtent` that maps an extend *eid* at offset *boff* of mfile *mfid*. Figure 4.4 shows an example log fragment where

multiple storage-object log entries are nested within a file-system log record for creating a file.

The ability to log operations from both storage objects and the file system enables the TFS server to benefit from work done at the client. Instead of having the server perform all the work needed for a file-system operation, the client may decompose the operation into individual storage-object operations that the server validates and performs. The client may guide the server by providing hints, which the server can use to skip work that otherwise it would have to do. For example, Figure 4.5 shows the log fragment of a file write that requires allocating new storage. The client logs the file-system operation, and then logs each individual storage extent it pre-allocates (`createExtent`) and wishes to attach to the file (`addExtent`). This allows the client to write the data to the extents without involving the server. When the client finally ships the log to the server to apply the changes, the server only has to validate each allocation and attach each extent to the file rather than having to allocate storage, write the data, and then attach the extents to the file. Operation `createExtent` also includes a hint to the free-set slot containing the extent, which the server can use to quickly validate that the file write uses a free extent.

The server validates a client's updates before applying them to ensure they maintain invariants. When a logged file-system operation comprises multiple storage-object operations, the server verifies that these operations as a whole maintain the storage-object invariants listed in Tables 4.3 and 4.4. For any operation that modifies an object, the server also verifies that the client holds necessary locks and permissions. Recent work has shown that such file-system integrity constraints can be specified as inference rules and enforced online using fast local checks on the data being modified [66]; we take a less automated approach but future work could consider incorporating their inference rule engine into our system. We implement our invariant checkers as state machines. For each storage-object operation, we provide a checker that parses a log record, validates that the record has a valid structure, corresponds to a known operation, and that the operation maintains invariants. The invariant check is implemented using pre- and post-condition checks that are derived from the invariant. File systems construct a checker for each file-system operation by connecting together individual

storage-object checkers into a state machine that validates a sequence of operations. Each machine state represents outstanding conditions that must be satisfied so as to accept the modifications as valid. Checkers transition the machine between states when an outstanding condition is satisfied or a new condition is introduced.

Putting it all together, reconsider the example in Figure 4.5. A checker for file write validates that each pair of `createExtent` and `addExtent` maintains invariants 4.2 and 4.3. Invariant 4.3 is validated by checking that `createExtent` properly allocates the free extent needed by `addExtent`, and vice versa.

Crash Recovery

The TFS server uses write-ahead logging implemented using a redo log to atomically perform multiple metadata updates. The server first logs each metadata update, flushes the log, and issues a fence to ensure following writes are ordered after the log writes. It then writes and flushes metadata using wlflush. In case of a crash, the TFS server can recover by replaying the log of metadata updates. The server does not need to reacquire locks as updates were written and ordered in the log with locks held.

Free Space Management

The TFS service implements a buddy storage allocator [105, 106] to create extents out of a partition. Clients do not allocate storage directly through the buddy allocator. Instead, libFS pre-allocates a pool of 1000 collections, 1000 mFiles, and 1000 extents to avoid contacting the service for create or append operations. The service maintains a collection that tracks the pre-allocated objects owned by each client to prevent memory leaks.

4.3 File Systems Interfaces on Aerie

A major goal of Aerie is to provide a substrate for flexible file-system design. To demonstrate this capability, we implemented two file system interfaces. The first, PXFS, shows how to use the storage abstractions to implement a POSIX-style file system interface for compatibility

with existing code. The second one, KVFS, shows how to optimize the interface for a specific workload.

4.3.1 PXFS: POSIX-style File System

PXFS provides most POSIX semantics for files and directories, including moving files across directories, retaining access to open files after its permissions change or it is unlinked, and permission checks on the entire path to a file. It does not provide asynchronous update of timestamps or predictable file-descriptor numbers.

Storage Objects. We implement POSIX storage objects directly with mFiles and collections. Files are mFiles with page-size extents and directories are a collection mapping file names to the object IDs of files and directories. A root collection holds the root directory. Because the storage layer provides most of the mechanisms to access file data, the file and directory implementation is small. Since clients cannot directly modify storage objects, PXFS creates a temporary shadow object in the client when modifying metadata of an object. The shadow reflects the changes done locally by the client. It enables the client view an object image that is consistent with the operations the client has performed before sending the metadata updates to the TFS server. For example, when writing to a file, the client creates a shadow file object where the client buffers its metadata updates. The client bypasses file reads through the shadow so that it accesses writes it has performed.

Integrity. We construct integrity checks for file-system operations as described in Section 4.2.4. For invariant 4.7 we define noCycle to check that the directory collection added is not already included in another directory collection. This is a conservative check that similarly to POSIX ensures there are no cycles by preventing hard links to directories.

Naming. We implement a hierarchical namespace by organizing the directory collections into a tree. To create a file within a directory, a client creates an mFile, acquires a read/write lock on the directory's collection, and then inserts the name and mFile's object ID. To atomically

rename a file between directories, PXFS acquires read/write locks on old and new directory collections, inserts a name/file ID pair in the new destination and removes the name/file ID from the old collection. Since acquiring multiple locks, may lead to deadlock, we acquire all the locks in advance and release all locks if the TFS revokes a lock. Since acquiring read/write locks on collections forces other clients that hold locks on the collection to send their modifications to the service, directory updates that happen near the root directory may be slow.

PXFS supports both absolute and relative path resolution. Absolute paths are resolved starting at the root by recursively acquiring a read-only lock on each directory collection until the name is resolved. Relative paths are resolved starting at the working directory by recursively acquiring locks up or down the directory hierarchy to prevent concurrent renames of directories higher up the tree.

File sharing. PXFS supports concurrent file access. When a client opens a file, it acquires a lock on the file's mFile, which it holds until it closes the file. To allow files to be unlinked while open, the PXFS TFS service maintains a table of open files that are not locked. When another client requests the lock on an open file, clients with the file open notify the service that the file is open when releasing the lock. The service then adds the file into a collection of currently open files. The client can still obtain explicit locks on the mFile to read or write data, and when the client terminates or notifies the service that it has closed the file, the service reclaims the file's memory. This design guarantees the client can directly access the file even if other clients unlink or rename it.

Permission changes are handled similarly: memory protection is updated synchronously when the permissions change, but processes with the file open notify the service. They can then access the file through the service over RPC. This approach to sharing is similar to Sprite's support for consistent read/write sharing [143], which reverts to sending requests to the server when there are conflicting concurrent accesses to a file. As POSIX specifies that permissions are enforced along the path to a file (Windows by default does not), PXFS updates the protection on all objects underneath a directory when its permissions change.

Discussion. With this design, read-only access to files only communicates with the TFS service to acquire locks, and if there are no conflicting accesses, a coarse grained lock high in the file system tree suffices. The client can write to file data locally, including writing new data to files, but must communicate with the service for metadata changes such as creating or appending to a file.

We found that supporting POSIX semantics increases the complexity of the implementation. For example, to retain open files that have been unlinked, clients must communicate with the server to indicate when files are opened or closed. While we chose to provide this feature to support applications that depend on it, the performance cost when files do not need to be cached may be excessive.

4.3.2 KVFS: Key-Value File System

In order to demonstrate how an application can use Aerie's facilities to improve performance, we designed KVFS to provide a (i) simple storage model and (ii) a key-value store interface targeting applications that store and share many small files in a single directory, such as a web-proxy cache, an email client or wiki software. For example, a web-proxy cache that keeps a file per cached item can use KVFS to efficiently store and access cached items. At the same time an administrator can continue accessing the cache through PXFS, which provides backwards compatibility to her standard auditing scripts and utilities.

With KVFS, clients have a shared consistent view to files through a flat key-based namespace and access files through a simple put/get/erase interface. In addition, all files have the same permissions. In contrast to PXFS, KVFS does not support POSIX semantics, such as hierarchical namespace, unlinking or renaming open files, and multiple names for a file.

KVFS files are implemented with mFiles containing a single extent holding the entire file contents. The mFiles store no other metadata, such as permissions or access time. The file system does not have a hierarchical namespace, so all files are stored in a single collection that maps file names to mFiles. Thus, KVFS and PXFS use the *same memory layout* and differ in the policies the interface layer uses to allocate and synchronize data.

We enable scalable concurrent access to the flat key-based namespace through hierarchical locks. A single lock covers the whole collection and multiple locks under the single lock cover the extents that comprise the hash table of the collection. Each extent's lock also covers the files linked from the key-value pairs stored in the extent. Operations acquire the single collection lock in intent mode, and then acquire the lock covering the extent where the key-value pair is stored. Insert and delete operations acquire a read/write lock while lookups acquire a read lock. When insert or delete cause a rehash of the table, the rehash operation acquires the single lock covering the whole collection in read/write mode.

With this design, a client can operate almost entirely without communication. It can batch requests to pre-create file objects and allocate extents, and then commit groups of files at once. Furthermore, the get/put interface opens a file and returns its data in a single operation, which removes the need to maintain state about open files in memory. An alternative model to KVFS would be to implement a key-value store as a single large file, KVFS, in contrast, enables mutually distrustful programs to concurrently access and update files, such as for indexing or backup/restore.

4.4 Evaluation

The goal of Aerie is flexibility and performance. We evaluate performance with a mix of micro- and macrobenchmarks and compare against traditional and user-mode file systems. In addition, we evaluate the benefits of specializing file system design to a workload.

While we have limited experience building file systems for Aerie, the PXFS and KVFS file systems demonstrate the value of its layered architecture. Table 4.2 gives the size of each file system. For comparison, the ext3 file system in the Linux kernel is 11,663 lines, and the user-mode implementation for Fuse is 18,916 lines. KVFS, with reduced functionality, is only 340 additional lines of code yet provides synchronized access to files. While neither KVFS nor PXFS are as full-featured as ext3, their small size demonstrates the benefit of Aerie's storage abstractions to flexible file-system design.

4.4.1 Methodology

We performed our experiments on a at 2.4GHz Intel Xeon E5645 six-core (twelve thread) machine equipped with 48GB of DRAM running x86-64 Linux 3.2.2 kernel. For all our experiments we report averages of at least five runs.

Storage-class Memory. We follow the methodology described in Section 3.5.1 to emulate SCM using DRAM by adding delays to model SCM's performance. All tests add 150ns of extra latency.

Workloads. We compare Aerie against three Linux file systems: RamFS, ext3 and ext3 with FUSE. RamFS uses the VFS page cache and dentry cache as an in-memory file system. We modified RamFS to introduce delays when writing data and metadata in the caches to account for the extra delay of PCM. RamFS does not provide any consistency guarantees against crashes; thus it serves as a best-performing kernel-mode file system. To compare against file systems that provide crash consistency, we constructed an emulator, *SCM-disk*, for a PCM-based block device. Based on Linux's RAM disk (`brd` device driver), SCM-disk introduces delays when writing a block to limit the effective bandwidth. We mount an ext3 file system on SCM-disk. In addition, we also configured a user-mode version of ext3 using FUSE [175] that writes data to SCM-disk. We use 16GB memory partition for all four configurations.

We wrote our own microbenchmarks that stress specific file operations. For application-level workloads, we use a modified version of FileBench [5] that calls through libFS rather than system calls. Unless otherwise specified, workloads are single threaded.

4.4.2 Microbenchmark Performance

Individual operations. A prime motivation for Aerie is that direct access to storage can make user-mode file systems as fast as ones in the kernel. We evaluate the latency of common file-system operations. The sequential tests operate on a 1GB file in 4KB blocks, and the random workloads randomly access 100MB out of a 1GB file in 4KB blocks. Open/create/delete are

Benchmark	Latency (μs)			
	RamFS	**ext3**	**ext3-FUSE**	**PXFS**
Sequential read	0.77	0.83	3.5	0.7
Sequential write	2.1	1.9	18.4	1.5
Random read	1.2	4.5	25.5	1.2
Random write	1.4	3.8	20.8	1.6
Open	2.0	4.7	127.0	3.7
Create	9.2	113.4	2044.5	13.4
Delete	3.1	11.6	227	2.7
Append	5.4	7.5	25.3	4.0

Table 4.5: Latency of common file system operations. All read/write operations use a 4096-byte buffer.

measured by opening/creating/removing 1024 4KB files. Because Aerie batches updates, we report average latency.

Table 4.5 shows the latency of common file system operations on PXFS, RamFS, and both ext3 versions. As expected RamFS performs consistently better than ext3 except for sequential write. Writes in RamFS are performed directly to SCM whereas in ext3 they are staged in RAM. PXFS performs close to RamFS for all operations but `create` and `open`, where PXFS latency is 45% and 85% higher respectively. Opening a file takes longer for PXFS because pathname resolution walks the persistent directory structure for each path component, and creates a shadow object to buffer metadata updates. Of the 3.7μs to complete an open call, 0.85μs is spent in lookup and 1.5μs in creating a shadow. In contrast, RamFS already has the objects in memory so it only pays the overhead of looking up directory entries in the dentry cache, which is highly optimized for lookups. Adding a path-name cache to PXFS or deferring shadow object creation until the first write can reclaim much of the performance difference.

Compared with ext3 in the kernel, PXFS is between 15% to 90% faster (average 35%) for all operations. Open is faster for PXFS because ext3 has to bring the file into the inode cache. PXFS benefits by not calling into the kernel, which helps all writes and random reads, and by batching metadata updates for create, delete and append.

In comparison to Aerie's performance, user-mode performance with FUSE was between 5 to 21 times slower for read/write operations and between 10 to 150 times slower for metadata

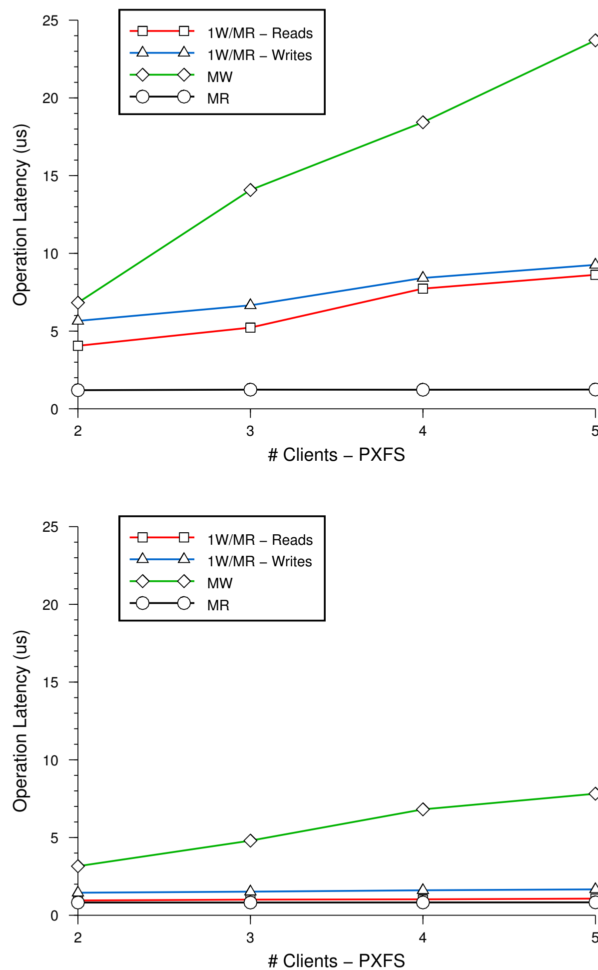


Figure 4.6: Performance of concurrent clients reading or writing a 4KB page of the same file. 1W - one writer, MW - multiple writers, and MR - multiple readers.

operations, largely due to the extra context switches and data copies necessary to invoke the file-system service. Aerie provides a flexible means of implementing file systems with high performance.

We separately measure the cost of changing file permissions. The TFS server asks the SCM manager to change memory protection on pages storing the file with the `scm_mprotect_extent`. If a page has been referenced and is in a page table, the SCM manager shoots down the page from the TLB and invalidates its page table entries. Changing protection takes 3.3µs per page that has been referenced, most of which is TLB shootdowns time. For large files, it may be

Benchmark	Latency (μs)			
	RamFS	ext3	ext3-FUSE	PXFS
Fileserver	198 (218)	424 (764)	5324	270 (292)
Webserver	143 (132)	158 (148)	2728	189 (195)
Webproxy	90 (66)	140 (130)	234	86 (64)

Table 4.6: Average latency and 95-percentile latency (in parentheses) to complete one workload iteration.

faster to flush the entire TLB.

Sharing. Due to the lack of workloads that exhibit sharing between applications, we measure instead the impact of sharing on our system through a stress microbenchmark. Figure 4.6 shows the latency of reading or writing a 4KB page of the same file shared between multiple concurrent clients in PXFS. As expected, multiple writers exhibit poor performance due to heavy lock contention, but a single writer with multiple readers degrades read and write performance only slightly. Running the same microbenchmark on ext3 shows similar behavior but with better absolute performance than PXFS due to in-kernel shared memory locking rather than distributed locking.

4.4.3 Application Workload Performance

We evaluate application-level performance with three FileBench profiles (*Fileserver*, *Webserver*, and *Webproxy*) to exercise different aspects of the file system. The Fileserver workload emulates file-server activity and performs sequences of creates, deletes, appends, reads, and writes. The Webserver workload performs sequences of open/read/close on multiple files and appends to a log file. Webproxy performs sequences of create/write/close, open/read/close, and delete operations on multiple files in a single directory plus appends to a log file. Each workload is broken up into individual iterations, and we report the latency of an iteration. The Fileserver and Webserver benchmark use 10,000 files, mean directory width of 20, and a 1MB I/O size. The mean file size was 128KB for the Fileserver and 16KB for the Webserver. The Webproxy benchmark was run with 1000 files, mean directory width of 1500, mean file size of 16KB, and 1MB I/O size.

Table 4.6 shows the average latency to complete one workload iteration. Compared to RamFS, PXFS is 35% slower for Fileserver and Webserver, but matches the performance of Webproxy. The microbenchmark results in Table 4.5 explains much of these results. Additionally, the Fileserver workload uses larger writes (128KB) than the microbenchmarks (4KB), which amortize the cost of entering the kernel and lead to 25% better performance than PXFS.

Webserver is a read-mostly workload (opens/read/close) to small files. RamFS performs better because of the lower latency to open a file (60% lower) due to in-memory caching of directory entries. In contrast, PXFS has no caching and must re-walk persistent structures on every access. PXFS matches the performance of RamFS on Webproxy because there is only a single directory lookup.

Compared to ext3, which in contrast to RamFS provides crash consistency, PXFS achieves 36% and 39% lower latency for the Fileserver and Webproxy workloads. Both workloads have a large fraction of file creates and deletes, writes, and random access, for which PXFS is substantially faster than ext3. For Fileserver, the large performance improvement comes from PXFS's better write performance for writes (103μs vs. 220μs) and 70% faster deletes (19μs vs. 62μs). The Webserver workload, though, is almost entirely sequential data reads and performs worse on PXFS for the same reasons that PXFS performs worse than RamFS.

A large benefit for PXFS comes from batching, which is not possible in ext3 because the kernel releases locks before returning to user mode. We found the average optimum batch size for our workloads to be 8MB. As shown in Table 4.6, batching for PXFS affects latency variance only slightly, with 95-percentile latency only slightly higher than average latency. The exception is Webproxy where the 95-percentile is lower due to a single outlier pulling the average up. Note that the other file systems show similar outlier effects, even without batching.

4.4.4 Client and Server Scalability

The preceding results evaluated single-threaded performance. First we evaluate the effect of having multiple threads in the client. Figure 4.7 shows throughput (workload iterations per

Benchmark	Throughput (workload iterations/s)			
	1	2	4	6
Fileserver	3974	8532	12131	18190
Fileserver+Webproxy	N/A	6217	11784	16200

Table 4.7: Throughput performance of a multiprogrammed workload with increasing client processes.

second) for our three workloads as we vary the number of threads in a single client process. For Fileserver, PXFS achieves better scalability than ext3 and doubles throughput when going from 1 to 6 threads. For the other two workloads, Webserver and Webproxy, PXFS throughput does not increase because of single-lock bottlenecks. (1) For Webserver, we see increased contention (20% of total runtime) for an internal lock in the storage object implementation. (2) For Webproxy, we see contention (22% of total runtime) for the lock covering the single directory. With extra effort, both locks can be split into multiple fine-grained locks to remove this contention.

RamFS presents near-linear scalability for all three workloads primarily because it does not have to write to a journal, which becomes a scalability bottleneck for ext3 (5% for 1 thread vs. 30% for 4 threads of total run time is spent in journaling). RamFS also benefits from scalable RCU synchronization for directory lookups.

We evaluate the scalability of the TFS server by running two multiprogrammed workloads: (1) multiple single-threaded Fileserver instances, and (2) Fileserver+Webproxy, which runs an equal number of Fileserver and Webproxy instances. We do not consider Webserver; as a read-mostly workload it does not put much pressure on the server. We configured each client to operate in a different directory to avoid contention between clients due to locking.

Table 4.7 shows the aggregate throughput of both tests and suggests that both workloads can scale well (3x speedup for 4 clients). This is because multiple threads in the TFS server can perform metadata updates concurrently under different parts of the namespace due to hierarchical locks. The server CPU utilization increases from 24% for 1 client to 72% for 4 clients.

4.4.5 Workload-Specific Performance

A key motivator for Aerie is the ability to create workload-specific file system interfaces. We compare the performance of KVFS with a get/put interface in a single directory against PXFS for the Webproxy workload, whose usage fits the KVFS interface. We modified the Webproxy workload by converting the create-write-close file sequence to a put operation, open-read-close file to a get operation and delete to an erase operation. We convert the append to a get/modify/put sequence.

Figure 4.7 shows the performance of KVFS for the Webproxy workload. For a single thread, KVFS is 86% faster than RamFS and 79% faster than PXFS. With six threads, it is 30% faster than RamFS and 415% faster than PXFS. With a single thread, the biggest benefit comes from using a get/put interface instead of open/read/write/close. With the standard interface, PXFS must create a temporary in-memory object representing an open file and record the file offset on every read. With get/put, KVFS can locate the file in memory and copy it directly to an application buffer. With multiple threads, the performance benefit comes from using multiple locks within a single directory, which alleviates the scalability limitations of PXFS. In addition, data access is faster with KVFS because it stores files in a single extent rather than using a radix tree of multiple extents. Thus, getting or putting data is a single `memcpy` operation.

4.5 Summary

New storage technologies promise high-speed access to storage directly from user mode. The existing file-system architecture, where a shared kernel component mediates all access to data, unnecessarily limits performance both by interposing on requests and complicating file system implementation. The Aerie architecture represents a new design targeting storage-class memory, and reduces the kernel role to just multiplexing physical memory. As a result, applications can achieve high performance by optimizing the file-system interface for application needs without changes to complex kernel code.

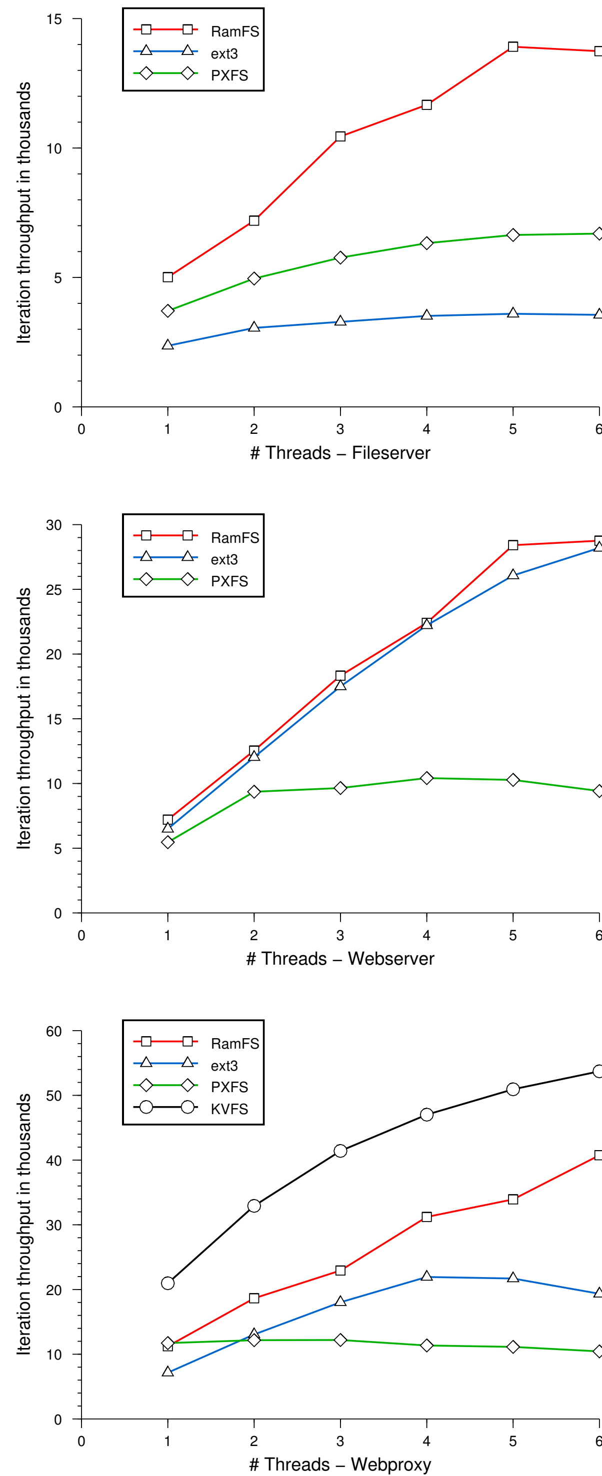


Figure 4.7: Throughput performance (workload iterations per second) as a function of the number of threads in a single client.

5 CONCLUSIONS

Emerging storage-class memory (SCM) devices provide the interface of memory but the persistence of disks. SCM promises low-latency storage, which can benefit modern applications, ranging from smartphone applications [101] to large-scale web applications such as social-network graphs [141]. However, existing operating-system interfaces fail to expose the full capabilities of SCM as they are designed for a strict bifurcation of devices into volatile memory and persistent storage. This thesis has made a first step towards revisiting the system interface to storage in light of SCM. Below we summarize the contributions of this thesis, share some lessons learned while working on this dissertation, and discuss future directions that could be further investigated.

5.1 Summary

We presented two proposals that provide user-mode programs direct access to the low-latency durability of SCM. Direct access bypasses system software layers, thus enabling persistence at the speed of SCM rather than the speed of OS.

Our first proposal, Mnemosyne, provides a simple interface for programming with persistent memory. In Mnemosyne, programmers declare global persistent data with the pstatic keyword or allocate it dynamically. Mnemosyne also provides primitives for directly modifying persistent variables and supports crash-consistent updates through a lightweight transaction mechanism. With persistent memory, programmers can make common in-memory data structures, such as trees, lists, and hashes, persistent without first converting them to a serialized format. In tests emulating the performance characteristics of forthcoming SCMs, we showed that Mnemosyne can persist data as fast as 3 microseconds. Furthermore, after converting three applications, OpenLDAP, Tokyo Cabinet, and Memcached, to use persistent memory, we found that the performance of moving existing in-memory structures to persistent memory is 35-1400% faster than Berkeley DB's block-optimized storage or flushing the whole structure to a file. Overall, these results demonstrate that programmers can create a single data structure,

optimized for memory, rather than designing separate in-memory and update-optimized persistent representations.

Our second proposal, Aerie, exposes a file-system interface to SCM using user-mode libraries that access file-system data directly through memory. While a file system does not provide the fine-grain persistence of persistent memory, it does enable interoperability between applications through a common interface to storage. At the same time, implementing the file system as a user-mode library enables applications to optimize the file system to their specific needs; an application may access storage through an interface or data layout suited to its workload. We showed that Aerie is generic enough to build a POSIX-style file system that offers applications backward compatibility and achieves performance similar to or better than a kernel implementation. We found that a POSIX-style file system on Aerie performs 35% better than ext3 over SCM and only 15% worse than a kernel file system without crash consistency. We also demonstrated the value of application-specific file systems through a specialized file system that reduces a hierarchical file system abstraction to a key/value store with fewer consistency guarantees but lower latency. The specialized design enabled by Aerie improves performance by 30-86% over a kernel file system for its workload. Overall, these results demonstrate that with Aerie applications can improve file-system performance by customizing storage layouts and interfaces.

5.2 Lessons Learned

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

Software plays a new role in storage. Our experience building software abstractions and interfaces to storage-class memory can be distilled in the following lessons:

- In traditional block-based storage devices, such as disks, the dominant cost was the latency getting data off the device so software overhead was not a major a concern. In fact, software has played a positive role in abstracting device complexity and making

the system faster through techniques such as buffering data in DRAM and intelligent prefetching. In contrast, with fast SCM, software can be a major contributor of overhead to access latency.

- Direct access to SCM can eliminate software overheads, but direct access alone is not sufficient as it lacks common features we expect from storage, such as naming, crash consistency, and sharing semantics. Thus, software will continue playing a role in providing these features, but it has to transform from thick, generic storage stacks to lightweight, flexible, and composable mechanisms so that applications can compose the storage semantics they need and minimize the performance cost they pay.
- While implementing storage-related mechanisms in user mode avoids the overhead calling into the kernel, the real performance benefit comes from giving applications the flexibility to customize storage to their needs. For example, we found that implementing POSIX in user mode did not give huge performance benefits over its kernel-mode counterpart. This is because we ended up implementing similar functionality to support POSIX semantics, such as in-memory inodes to keep track of open files. However, a specialized file system that provided a key-value interface on the same layout of the POSIX file system gave significant performance improvement.

Current hardware lacks functionality. We identify necessary functionality and several useful features that current commodity hardware is missing:

- Current processor primitives do not guarantee completion. Current memory systems acknowledge the completion of a data write as soon as the memory controller buffers it, that is, before the write actually hits the SCM cell and becomes durable. Therefore, a failure that happens after the acknowledgement but before the memory controller performs the updates to SCM causes the update to be lost. One approach to addressing this problem would be to extent the memory controller to acknowledge the completion of a write only after the write hits the SCM cell. However, this approach could have a negative impact on performance when used with slow SCM as it would increase

completion time. Another approach would be to make the buffer persistent by using either a supercap that provides enough energy to complete buffered writes, or STT-MRAM that makes the buffer persistent and allows the controller to complete buffered writes upon restart.

- While current processor hardware primitives provide the necessary functionality for flushing data out to memory, they may negatively impact performance. For example, the x86 `clflush` instruction writes a cache line back to memory and then invalidates it. Invalidation is unnecessary and may affect performance negatively if data is accessed after it is flushed to memory. Processor hardware could be extended to provide an instruction that performs only the write back.

- While the protection mechanism provided by current virtual memory hardware is sufficient for correctly enforcing file system permissions, it not as flexible and effiecient as we would like it to be. For example, x86 provides only three protection rights: no access, read-only access, and read-write access. This limits the permission rights that can be efficiently enforced directly by hardware. For example, x86 does not provide write-only protection so we revert to software to enforce write-only access to files. As another example, changing the permission rights of a file is not currently efficient as x86 requires separately changing the protection rights of each page that comprises the file. Other protection mechanisms that associate a set of pages with the same protection right, such as Itanium's protection keys [72], could be more efficient.

Caching may still be relevant but in a different form. The low-latency access of SCM encouraged us to not incorporate caching in our designs. In retrospect, caching may still be useful in improving access latency but not in its traditional form. Since SCM's access latency is higher than DRAM, regular caching of SCM pages in DRAM may be beneficial. However, we see a bigger benefit by caching in DRAM efficient data structures that are derived from their persistent counterparts. Such volatile structures can provide an efficient access path to persistent state and data. For example, Mnemosyne's persistent-heap allocator maintains

a simple persistent bitmap for tracking allocated and freed memory. In addition, it keeps volatile linked lists of free memory blocks in DRAM, which it derives from the persistent bitmaps, in order to speed up allocation. Aerie file systems could benefit from a similar form of caching. For example, namespace lookups in PXFS could benefit from a volatile index structure that is derived from the persistent directories.

Fast-storage shifts bottlenecks. As storage becomes faster, the performance gap between storage and other system components, such as CPU, memory, network and storage, is reduced. Thus, storage will become less of a bottleneck, and the bottleneck may shift to other less efficient parts of the application. For example, in the OpenLDAP directory service we found that for a write-intensive workload fast storage moved the bottleneck from the storage back end to the request-processing front end.

Locality still matters. In contrast to disks where geometry affects access latency due to seek and rotational delays, access latency in SCM does not depend on geometry. One would think then that with SCM random access will have constant latency and locality will be less relevant. We expect that random access will still have some variations because of architectural factors. For example, architecting SCM in multiple banks [115] increases throughput, which can benefit sequential accesses; a feature we exploit in our log design. Other architectural factors that affect access latency include caches and TLBs. Thus, we expect that locality will continue playing a role in the design of storage systems for SCM.

Hand-coded locking complicates recovery. In an early design of the transaction runtime system of Mnemosyne we considered memory transactions that provided only atomicity and durability (AD), and left isolation to the programmer similarly to other systems such as LRVM [115] and Rio Vista [120]. Our experience using AD memory transactions, where we hand-coded isolation using mutex locks, showed that this approach was quite error prone and complicated recovery. In contrast to LRVM and Rio Vista, which use a single shared log, Mnemosyne uses multiple per-thread logs for better scalability. Per-thread logging

complicates recovery because there is no longer a single ordering point that orders transactions like a single shared log. Therefore, we need a mechanism that upon recovery reconstructs the order transactions executed. Mnemosyne deals with this problem by logging the commit token number of each transaction after making the transaction durable so that it can replay transactions in commit order upon recovery. Decoupling atomicity and isolation requires programmers maintain the invariant that isolation is released only after logging the commit token, otherwise the order transactions executed and made durable may not correspond to the commit order. We initially hit this pitfall, and our hand-coded isolation acquired and released locks before transaction commit. Recovery could no longer rely on the commit order to properly replay per-thread redo logs. To properly recover, we had to expose to the programmer the internals of the transaction runtime system, which however would prevent independent evolution of the transaction system from application code. A solution we considered was to rely on transaction-safe locks [187], which delay releasing locks after the transaction commits. However, relying on a specific synchronization mechanism did not adequately delivered our goal of decoupling atomicity and isolation. We eventually abandoned AD transactions in favor of full-fledged ACID memory transactions, which are more programmer friendly.

References from persistent to volatile memory, while dangerous, can also be useful. While we do not generally encourage keeping pointers to volatile memory in a persistent region, we found this practice can be useful when a persistent data structure needs to include state that is logically volatile (*i.e.*, state that does not outlive the application). Such volatile state can be temporary such as statistics counters or state that is recreated by the application as in the case of OpenLDAP (Section 3.5.2). Because volatile state becomes stale after a restart, the programmer has to write recovery code that properly reinitializes persistent pointers to volatile state.

5.3 Future Work

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

Performance. In general, we see two interesting directions for improving performance. First, some SCM technologies have asymmetric performance. For example, PCM's read latency is faster than its write latency by at least an order of magnitude. An interesting direction then would be to explore data structures that perform some extra fast reads to save some slow writes. Second, since SCM's access latency is higher than DRAM, caching SCM pages in DRAM or creating a faster secondary access path in DRAM to data stored in SCM may be beneficial. More specifically, in the context of Aerie, we see another three interesting directions. First, an Aerie file system may map terabytes of data into a process. Current page tables for mapping this data can be large, which introduces large space overheads and also puts huge pressure on the TLBs. One approach in dealing with this problem would be to decouple mapping from protection. Since Aerie partitions are mapped to physically contiguous memory, mapping mechanisms other than page tables could be used, such as segmentation hardware, to reduce the amount of mapping information stored and accessed. Separating mapping from protection also enables exploring other perhaps more efficient protection mechanisms such as Itanium's protection keys [72]. Second, currently the smaller unit of protection is the page, which causes internal fragmentation for file-system objects smaller than a page. One approach to reducing fragmentation would be to group objects that have the same protection together in the same page. Third, through KVFS we demonstrated that matching the file system to the application can significantly improve performance. One interesting question is how much of the performance benefit could be achieved by having a get/put interface on top of the kernel file-system interface. Moreover, it would also be interesting to study and develop specialized file systems for other applications, such as a mail server or a web server.

Reliability. While direct access to SCM enables tremendous opportunities for low-latency flexible storage, it also poses important system reliability challenges. First, the wide memory interface to SCM makes stored data susceptible to accidental corruption such as wild writes. We see three potential solutions to this problem. A first solution would be to use page-level protection to enable write access to pages only when they are actively used. However, this approach would suffer from the high cost of changing protection, thus opening up the question of how to efficiently protect pages. A second solution would be the use of data structures resilient to faults [179, 63] and techniques for repairing corrupted data structures [190]. Finally, a third solution would be to hide SCM in the address space and provide access only through a managed language such as Java. This approach, however, raises several issues, such as how the garbage collector would have to interact and manage persistent metadata.

A second reliability challenge arises from attaching SCM to the memory bus. While this approach enables direct access, it also ties SCM to a single system, thus making data stored in SCM inaccessible in case of system failure. One approach to addressing this issue would be to replicate data to either a locally attached storage array or a remote node over the network (similarly to RAMCloud [141]).

Security. Direct access may leak stored data if the OS does not properly recycle pages between users (or processes). In contrast to the current system interface that mediates access to each SCM-page and therefore controls what content is visible, the techniques presented in this dissertation enable direct access to the contents of an SCM-page by memory mapping the page into a process. If the OS does not properly erase (*i.e.*, zero) the old contents of a page before reallocating and memory mapping a page, then the new process can see any data previously stored in the page by another user (or process). While erasing the old contents of the page solves the problem, it increases wear out of the page and also the time to free or allocate a page. Thus, other more efficient techniques may need to be devised.

Consistency. With current processors, performing crash-consistent updates requires flushing and ordering updates to SCM. Our tornbit log design showed that simple versioning

schemes can reduce the number of ordering constraints and therefore the number of dependent flushes that need to be done sequentially. One could perhaps envision taking this idea further, and explore other simple versioning schemes that at once reduce the number of ordering constraints and allow more generic updates than log appends.

5.4 Closing words

Upcoming storage-class memory technologies promise an exciting new way for applications to store persistent data at near-DRAM speeds. Within the framework of this thesis we have only taken the first steps in designing a storage architecture and system interface that unleash the full power of SCM to the application programmer. Future directions outlined above pave the way for further work in designing storage systems for storage-class memory.

[1] Memcached: A distributed object caching system. http://memcached.org.

[2] Sparse: A semantic parser for C. sparse.wiki.kernel.org.

[3] SQLite. http://www.sqlite.org/.

[4] Intel C++ STM compiler prototype edition 3.0, 2008.

[5] Filebench benchmark. http://sourceforge.net/apps/mediawiki/filebench, 2011.

[6] International technology roadmap for semiconductors. http://www.itrs.net, 2011.

[7] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[8] AgigaTech. Agigaram (tm) non-volatile system. http://www.agigatech.com/agigaram.php, 2012.

[9] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: a protoype phase change memory storage array. In *Proceedings of the 3rd Workshop on Hot Topics in Storage and File Systems*, June 2011.

[10] AMD, Inc. Software optimization guide for AMD64 processors. http://support.amd.com/us/Embedded_TechDocs/25112.PDF, 2005.

[11] AMD, Inc. Advanced synchronization facility: Proposed architectural specification, rev. 2.1. http://developer.amd.com/assets/45432-ASF_Spec_2.1.pdf, Mar. 2009.

[12] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[13] M. Atkinson and M. Jordan. Issues raised by three years of developing pjama: An orthogonally persistent platform for java. In *Proceedings of the 7th International Conference on Database Theory*, Jan. 1999.

[14] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, Nov. 1983.

[15] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, June 1996.

[16] M. Baker and M. Sullivan. The recovery box: Using fast recovery to provide high availability in the unix environment. In *Proceedings of the USENIX Winter Technical Conference*, Dec. 1992.

[17] P. R. Barham. A fresh approach to file system quality of service. In *Proceedings of the IEEE 7th International Workshop on Network and Operating System Support for Digital Audio and Video*, 1997.

[18] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Oct. 2009.

[19] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

[20] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, M. Tosi, R. Bez, R. Gastaldi, and G. Casagrande. An 8Mb demonstrator for high-density 1.8v phase-change memories. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, June 2004.

[21] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[22] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software - Practice & Experience*, 18(9):807–820, 1988.

[23] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. Technical report, Sun Microsystems, 2002.

[24] Boost C++ Libraries. Serialization overview. `http://www.boost.org/doc/libs/1_42_0/libs/serialization/doc/index.html`, Nov. 2004.

[25] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov. 2005.

[26] T. C. Bressoud, T. Clark, and T. Kan. The design and use of peristent memory on the DNCP hardware fault-tolerant platform. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2001.

[27] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4):449–464, July 2008.

[28] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[29] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of the 2010 ACM SIGMOD international conference on Management of data*, June 2010.

[30] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012.

[31] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Twenty-Sixth ACM Symposium on Principles of Distributed Computing*, Aug. 2007.

[32] A. Chang and M. F. Mergen. 801 storage: architecture and programming. *ACM Transactions in Computer Systems*, 6(1):28–50, Feb. 1988.

[33] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions in Computer Systems*, 12:271–307, Nov. 1994.

[34] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The rio file cache: surviving operating system crashes. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996.

[35] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Proceedings of 5th Biennial Conference on Innovative Data Systems Research*, Jan. 2011.

[36] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX conference on File and Storage technologies*, Feb. 2012.

[37] J. H. Choi, H. Franke, and K. Zeilenga. Enhancing the performance of openldap directory server with multiple caching. In *International Symposium on Performance Evaluation of Computers and Telecommunications Systems*, July 2003.

[38] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, J. K. B. Cho, D. K. Y. Oh, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Wang,

S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J. Yoo, and G. Jeong. A 20nm 1.8v 8gb pram with 40mb/s program bandwidth. In *Proceedings of the 2012 IEEE International Solid-State Circuits Conference*, Feb. 2012.

[39] Christopher Clark. C hash table. `http://www.cl.cam.ac.uk/~cwc22/hashtable/hashtable.c`.

[40] L. O. Chua. Memristor-the missing circuit element. *Circuit Theory, IEEE Transactions on*, 18(5):507 – 519, Sept. 1971.

[41] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011.

[42] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Oct. 2009.

[43] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe ram. In *Proceedings of the 15th International Conference on Very Large Data Bases*, Aug. 1989.

[44] J. Corbet. Fsyncers and curveballs (the firefox 3 fsync() problem). `http://lwn.net/Articles/283745/`, May 2008.

[45] M. Corporation. ReFS: Resilient file system. `http://msdn.microsoft.com/en-us/library/windows/desktop/hh848060(v=vs.85).aspx`, 2012.

[46] O. Corporation. Oracle berkeley DB database. http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html.

[47] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: a case study in the effectiveness of best effort hardware transactional

memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011.

[48] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jan. 2010.

[49] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[50] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: a new approach to improving file systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Dec. 1993.

[51] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, June 1984.

[52] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing*, Sept. 2006.

[53] E. Doller. Phase change memory and its impacts on memory hierarchy. `http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf`, 2009.

[54] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie. Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems. In *Proceedings of the 2009 Conference on High Performance Computing Networking, Storage and Analysis*, Nov. 2009.

[55] Dovecot. Mailbox formats. `http://wiki.dovecot.org/MailboxFormat/`.

[56] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*, June 2009.

[57] U. Drepper. Futexes are tricky. `www.akkadia.org/drepper/futex.pdf`, 2005.

[58] N. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt. MRAMFS: A compressing file system for non-volatile RAM. In *Proceedings of thje IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Oct. 2004.

[59] F. Eskesen, M. Hack, A. Iyengar, R. King, and N. Halim. Software exploitation of a fault-tolerant computer with a large memory. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing Systems*, June 1998.

[60] Everpsin. Everspin debuts first spin-torque mram for high performance storage systems. `http://www.everspin.com/PDF/ST-MRAM_Press_Release.pdf`, Nov. 2012.

[61] Facebook. More details on today's outage. `http://www.facebook.com/note.php?note_id=431441338919`, Sept. 2010.

[62] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008.

[63] I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient search trees. In *Proceedings of the 18th annual ACM-SIAM symposium on Discrete algorithms*, Jan. 2007.

[64] I. I. T. R. for Semiconductors. Process integration, devices, and structures (pids). `http://www.itrs.net/Links/2011ITRS/2011Chapters/2011PIDS.pdf`, 2011.

[65] R. F. Freitas and W. W. Wilcke. Storage-class memory: the next storage system technology. *IBM Journal of Research and Development*, 52(4):439–447, 2008.

[66] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX conference on File and Storage technologies*, Feb. 2012.

[67] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. Pcmlogging: reducing transaction logging overhead with pcm. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, Oct. 2011.

[68] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[69] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, Oct. 2003.

[70] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[71] G. A. Gibson, D. Rochberg, J. Zelenka, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, and E. Riedel. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, June 1997.

[72] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser. Itanium: a system implementor's tale. In *Proceedings of the 2005 USENIX Annual Technical Conference*, June 2005.

[73] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Dec. 1989.

[74] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, 1978.

[75] J. Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the 7th International Conference on Very Large Data Bases*, Sept. 1981.

[76] J. Gray, P. Mcjones, M. Blasgen, B. Lindsay, R. Lorie, and T. Price. The recovery manager of the system r database manager. *ACM Computing Surveys*, 13:223–242, 1981.

[77] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[78] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Readings in database systems*, pages 94–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[79] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer Technical Conference*, June 1994.

[80] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and F. Jinpeng Wei. Software persistent memory. In *Proceedings of the 2012 USENIX Annual Technical Conference*, June 2012.

[81] X. Guo, E. Ipek, and T. Soyata. Resistive computation: avoiding the power wall with low-leakage, stt-mram based computing. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, June 2010.

[82] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[83] R. Haring. The IBM Blue Gene/Q compute chip. Hot Chips 23, Aug. 2011.

[84] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Morgan & Claypool Publishers, 2010.

[85] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[86] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the i/o behavior of apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating System Principles*, Oct. 2011.

[87] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[88] HIPEAC. Gcc for transactional memory. `http://www.hipeac.net/node/2419`.

[89] M. Hirabayashi. Tokyo cabinet: a modern implementation of DBM. `http://1978th.net/tokyocabinet/`, 2010.

[90] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter Technical Conference*, Dec. 1994.

[91] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, 2005.

[92] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6:51–81, Feb. 1988.

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

[94] Intel Corp.,. *Intel architecture instruction set extensions programming reference*.

[95] Intel Corp.,. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 1: Basic Architecture*, Jan. 2011.

[96] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010.

[97] A. M. Joshi. Adaptive locking strategies in a multi-node data sharing environment. In *Proceedings of the 17th International Conference on Very Large Data Bases*, Sept. 1991.

[98] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon. FRASH: Exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Transactions on Storage*, 6:3:1–3:25, Apr. 2010.

[99] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, Oct. 1997.

[100] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *Electronic Computers, IRE Transactions on*, EC-11(2):223 –235, Apr. 1962.

[101] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX conference on File and Storage technologies*, Feb. 2012.

[102] J. J. Kistler. *Disconnected Operation in a Distributed File System*. PhD thesis, Carnegie Mellon, 1993.

[103] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10:3–25, Feb. 1992.

[104] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the USENIX Summer Technical Conference*, June 1986.

[105] K. C. Knowlton. A fast storage allocator. *Communications of ACM*, 8(10):623–624, Oct. 1965.

[106] D. Knuth. The art of computer programming volume 1: Fundamental algorithms. Addison-Wesley, Reading, MA.

[107] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architecture support for single address space operating systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[108] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30:8–19, 2010.

[109] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proceedings of theVLDB Endowment*, 5(12):1790–1801, Aug. 2012.

[110] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Communications of ACM*, 34(10):50–63, Oct. 1991.

[111] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3:125–143, Mar. 1977.

[112] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[113] C. R. Landau. The checkpoint mechanism in keykos. In *Object Orientation in Operating Systems, 1992., Proceedings of the Second International Workshop on*, 1992.

[114] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[115] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[116] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[117] J. Lion. *Commentary on the Sixth Edition UNIX Operating System*. The University of New South Wales, 1977.

[118] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, June 1996.

[119] R. A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems*, 2(1):91–104, Mar. 1977.

[120] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, Oct. 1997.

[121] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[122] W. Magazine. Hp memristors will reinvent computer memory by 2014. `http://denalimemoryreport.com/2012/07/12/wired-magazine-hp-memristors-will-reinvent-computer-memory-by-2014/`, July 2012.

[123] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with anvil. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Oct. 2009.

[124] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[125] M. K. McKusick and G. R. Ganger. Soft updates: a technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.

[126] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - the unix file system check program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, Apr. 1986.

[127] Micron. Omneo P8P PCM 128-Mbit Parallel PCM. `http://www.micron.com/get-document/?documentId=5829&file=p8p_parallel_pcm_ds.pdf`, Mar. 2011.

[128] Micron. Mobile LPDDR2-PCM. `http://www.micron.com/products/multichip-packages/pcm-based-mcp`, July 2012.

[129] Microsoft Corp. Sql server 2008 books online: Memory management architecture: Buffer management. `http://msdn.microsoft.com/en-us/library/aa337525.aspx`.

[130] E. Miller, S. Brandt, and D. Long. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

[131] 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, 1992.

[132] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture*, Feb. 2006.

[133] I. Moraru, D. G. Andersen, M. Kaminsky, N. Binkert, N. Tolia, R. Munz, and P. Ranganathan. Persistent, protected and cached: Building blocks for main memory data stores. Technical report, Carnegie Mellon University, 2011.

[134] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, Aug. 1992.

[135] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[136] D. Narayanan and O. Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012.

[137] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for c/c++. In *Proceedings of the 23rd SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2008.

[138] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, Oct. 2005.

[139] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[140] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.

[141] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating System Principles*, Oct. 2011.

[142] Oracle. Btrfs: B-tree file system. `https://btrfs.wiki.kernel.org/index.php/Main_Page`.

[143] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *IEEE Computer*, 21:23–36, Feb. 1988.

[144] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE Symposium on High-Performance Computer Architecture*, Feb. 2011.

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

[146] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, June 1988.

[147] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.

[148] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, Oct. 2005.

[149] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[150] M. K. Qureshi, J. K. Michele, Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based. main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009.

[151] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[152] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[153] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1987.

[154] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: an operating system for a personal computer. *Communications of ACM*, 23:81–92, Feb. 1980.

[155] J. E. Richardson, M. J. Carey, and D. T. Schuh. The design of the e programming language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, July 1993.

[156] S. Riegel, F. Mellender, and A. Straw. Integration of database management with an object-oriented programming language. In *Advances in Object-Oriented Database Systems*, pages 317–322. Springer Berlin / Heidelberg, 1988.

[157] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of ACM*, 17:365–375, July 1974.

[158] D. A. Roberts. *Efficient Data Center Architectures Using Non-Volatile Memory and Reliability Techniques*. PhD thesis, University of Michigan, 2011.

[159] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris. Operating system support for persistent and recoverable computations. *Communications of ACM*, 39(9):62–69, 1996.

[160] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions in Computer Systems*, 10(1):26–52, Feb. 1992.

[161] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2006.

[162] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the USENIX Summer Technical Conference*, June 1986.

[163] S. Sardashti and D. A. Wood. Unifi: Leveraging non-volatile memories for a unified fault tolerance and idle power management technique. In *Proceedings of the 26th International Conference on Supercomputing*, June 2012.

[164] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Dec. 1993.

[165] M. Saxena, M. A. Shah, S. Harizopoulos, M. M. Swift, and A. Merchant. Hathi: durable transactions for memory using flash. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, 2012.

[166] D. Schneider. The microsecond market. *IEEE Spectrum*, 49(6):66–81, June 2012.

[167] M. D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. PhD thesis, Massachusetts Institute of Technology, 1972.

[168] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[169] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, May 2009.

[170] N. H. Seong, D. H. Woo, and H.-H. S. Lee. Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, June 2010.

[171] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010.

[172] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: good, fast, cheap persistence for c++. In *Proceedings of the 7th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 1992.

[173] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

[174] F. G. Soltis. *Inside the AS/400*. Duke Press, second edition, 1997.

[175] Sourceforge. FUSE:Filesystem in userspace. `http://fuse.sourceforge.net`, 2005.

[176] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, Sept. 2005.

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

[178] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, and S. J. Mullender. Experiences with the amoeba distributed operating system. *Communications of ACM*, 33(12):46–63, Dec. 1990.

[179] D. Taylor and J. Black. Principles of data structure error correction. *IEEE Transactions on Computers*, C-31(7):602 –608, July 1982.

[180] V. Technology. Arxcis-nv (tm): Non-volatile dimm. `http://www.vikingmodular.com/products/arxcis/arxcis.html`, 2012.

[181] S. M. Thatte. Persistent memory: a storage architecture for object-oriented database systems. In *OODS '86: Proceedings on the 1986 international workshop on Object-oriented database systems*, 1986.

[182] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, Oct. 1997.

[183] K. Tsuchida, T. Inaba, K. Fujita, Y. Ueda, T. Shimizu, Y. Asao, T. Kajiyama, M. Iwayama, K. Sugiura, S. Ikegawa, T. Kishi, T. Kai, M. Amano, N. Shimomura, H. Yoda, and Y. Watanabe. A 64mb mram with clamped-reference and adequate-reference schemes. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 2010.

[184] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, 1998.

[185] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX conference on File and Storage technologies*, Feb. 2011.

[186] C. Villa, D. Mills, G. Barkley, H. Giduturi, S. Schippers, and D. Vimercati. A 45nm 1gb 1.8v phase-change memory. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference*, Feb. 2010.

[187] H. Volos, N. Goyal, and M. M. Swift. Pathological interaction of locks with transactional memory. In *Proceedings of the Third ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Feb. 2008.

[188] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011.

[189] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[190] H. Wang, B. He, V. Prabhakaran, and L. Zhou. Crystal: The power of structure against corruptions. In *Proceedings of the Fifth Workshop on Hot Topics in System Dependability*, June 2009.

[191] S. J. White and D. J. DeWitt. Quickstore: a high performance mapped object store. *The VLDB Journal*, 4:629–673, Oct. 1995.

[192] J. William E. Snaman and D. W. Thiel. The VAX/VMS Distributed Lock Manager. *Digital Technical Journal*, 1(5):29–44, Sept. 1987.

[193] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[194] P. Wu, M. M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. W. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, M. F. Mergen, X. Shen, M. F. Spear, H. Wang, and K. Wang. Compiler and runtime techniques for software transactional memory optimization. *Concurrency and Computation: Practice and Experience*, 21(1):7–23, Jan. 2009.

[195] X. Wu and A. L. N. Reddy. SCMFS: A file system for storage-class memory. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2011.

[196] M. Yonkovit. Tokyo Tyrant – The Extras Part I : Is it Durable? `http://www.mysqlperformanceblog.com/2009/11/10/tokyo-tyrant-the-extras-part-i-is-it-durable/`, Nov. 2009.

[197] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Nov. 1987.

[198] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: a zfs case study. In *Proceedings of the 8th USENIX conference on File and Storage technologies*, Feb. 2010.

[199] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.