UNIVERSITY OF CALIFORNIA, SAN DIEGO

Providing Fast and Safe Access to Next-Generation, Non-Volatile Memories

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Joel Dylan Coburn

Committee in charge:

Professor Rajesh Gupta, Co-Chair
Professor Steven Swanson, Co-Chair
Professor Ranjit Jhala
Professor Bill Lin
Professor Geoff Voelker

2012

The dissertation of Joel Dylan Coburn is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2012

DEDICATION

To my loving parents. You are my inspiration.

And to the memory of Njuguna Njoroge.

EPIGRAPH

The secret of life, though, is to fall seven times and to get up eight times.

—Paulo Coehlo, *The Alchemist*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am thankful for the support of many wonderful people during the long journey leading me to this dissertation.

This thesis is the product of the guidance of my advisers Steven Swanson and Rajesh Gupta. I am thankful for their patience and constant encouragement during this journey. Their mentorship both in how to do research and how to plan a career has been invaluable. I am thankful for all the opportunities that have come my way as a result of their hard work and dedication to graduate students like me.

I would like to thank my committee members for their feedback and guidance throughout this process. Thanks to Ranjit Jhala for his role in the NV-heaps work and teaching me so much about programming languages. Also, I am indebted to Geoff Voelker for his mentorship and friendship throughout my Ph.D. He introduced me to the best runs in San Diego.

I would like to thank the members of the Non-Volatile Systems Lab (NVSL) and the Microelectronic Embedded Systems Lab (MESL). My Ph.D. is the product of hard work by many different people, so I believe this is as much their success as it is mine. I am grateful to Trevor Bunker for his outstanding work on the MARS project. I wish to also thank Adrian Caulfield, Laura Caulfield, Ameen Akel, Arup De, Todor Mollov, and the rest of the NVSL.

I wish to heartily thank those at UC San Diego, Stanford, NEC Labs, and Xilinx. I have had some wonderful friends and mentors along the way. I find it too difficult to name them all, so I'll mention just a few but please know you are all very much appreciated. Thanks to Jack Sampson for interesting conversation and providing expert feedback. Thanks to Kaisen Lin for his encouragement and to the other members of MESL who always told me to keep going. Thanks to Nathan, Ganesh, Sravanthi, and many others for their friendship.

Finally, I must express many thanks to my parents. They have instilled in me a life-long passion for learning. Their love is immeasurable and continues to guide me each day of my life. Also, I would like to thank my sister for all her support over the years. Thank you everyone!

Chapters 2 and 3 contain material from "Providing Safe, User Space Access to Fast, Solid State Disks", by Adrian M. Caulfield, Todor I. Mollov, Louis Eisner, Arup De, Joel Coburn, and Steven Swanson, which appears in *ASPLOS '12: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems.* The dissertation author was the fifth investigator and author of this paper. The material in Chapters 2 and 3 is copyright ©2012 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

Chapters 1, 2, 4, 5, and 6 contain material from "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories", by Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which appears in *ASPLOS '11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems.* The dissertation author was the first investigator and author of this paper. The material in Chapters 1, 2, 4, 5, and 6 is copyright ©2011 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

Chapters 2 and 3 contain material from "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories", by Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson, wwhich appears in *MICRO-43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* The dissertation author was the third investigator and author of this paper. The material in Chapters 2 and 3 is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

Chapters 1, 3, and 6 contain material from "From ARIES to MARS: Reengineering Transaction Management for Next-Generation, Non-Volatile Memories", by Joel Coburn, Trevor Bunker, Rajesh K. Gupta, and Steven Swanson, which will be submitted for publication. The dissertation author was the primary investigator and author of this paper.

Chapters 1, 5, and 6 contain material from "Programming Language Support for Fast, Byte-Addressable, Non-Volatile Memory", by Joel Coburn, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which will be submitted for publication. The dissertation author was the primary investigator and author of this paper.

VITA

2001	B. S. in Computer Engineering Gonzaga University Spokane, Washington
2001-2004	Teaching assistant Stanford University
2003	M. S. in Electrical Engineering Stanford University
2004-2005	Research assistant NEC Labs Princeton, New Jersey
2005-2007	Systems engineer Xilinx, Inc. San Jose, California
2007-2012	Research assistant University of California, San Diego
2008	Internship Intellisis Corp. La Jolla, California
2012	Ph. D. in Computer Science (Computer Engineering) University of California, San Diego

PUBLICATIONS

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, Steven Swanson, “NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories”, *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2011.

Adrian M. Caulfield, Todor I. Mollov, Louis Eisner, Arup De, Joel Coburn, Steven Swanson, “Providing Safe, User Space Access to Fast, Solid State Disks”, *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2012.

Devi Sravanthi Yalamarthy, Joel Coburn, Rajesh K. Gupta, Glen Edwards, Mark Kelly, “Computational Mass Spectrometry in a Reconfigurable Coherent Coprocessing Architecture”, *IEEE Design and Test of Computers (D&T)*, 28:58–67, 2011.

Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, Steven Swanson, “Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories”, *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2010.

Adrian M. Caulfield, Joel Coburn, Todor I. Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, Steven Swanson, “Understanding the Impact of Emerging Non-Volatile Memories on High-Performance IO-Intensive Computing”, Memories”, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Novermber 2010.

Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, Jack K. Wolf, “Characterizing Flash Memory: Anomalies, Observations, and Applications”, *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2009.

Jayanth Gummaraju, Joel Coburn, Yoshio Turner, Mendel Rosenblum, “Streamware: Programming General-Purpose Multicore Processors Using Streams”, *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

Jayanth Gummaraju, Mattan Erez, Joel Coburn, Mendel Rosenblum, William J. Dally, “Architectural Support for the Stream Execution Model on General-Purpose Processors”, *Proceedings of the Sixteenth International Conference on Parallel Architecture and Compilation Techniques (PACT)*, September 2007.

Joel Coburn, Srivaths Ravi, Anand Raghunathan, Srimat Chakradhar, “SECA: Security-Enchanced Communication Architecture”, *Proceedings of the 2005 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, September 2005.

Joel Coburn, Srivaths Ravi, Anand Raghunathan, “Power Emulation: A New Paradigm for Power Estimation”, *Proceedings of the 42nd Annual Design Automation Conference (DAC)*, June 2005.

Joel Coburn, Srivaths Ravi, Anand Raghunathan, “Hardware Accelerated Power Estimation”, *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, March 2005.

ABSTRACT OF THE DISSERTATION

Providing Fast and Safe Access to Next-Generation, Non-Volatile Memories

by

Joel Dylan Coburn

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2012

Professor Rajesh Gupta, Co-Chair
Professor Steven Swanson, Co-Chair

Emerging non-volatile memory technologies such as phase change memory, spin-torque transfer memory, and the memristor, will provide many orders of magnitude decrease in latency compared to disk and flash memory, dramatic increases in bandwidth, and a byte-addressable interface similar to DRAM. These new memories will offer enormous performance gains and intuitive abstractions for storage, but fully realizing these benefits requires us to rid software of disk-centric optimizations, design decisions, and architectures that limit performance and ignore bottlenecks previously hidden by the poor performance of disk. The algorithms that storage and database systems use to enforce strong consistency guarantees are critical to performance, and current solutions are deeply tied to conventional disk technology. This dissertation addresses the problem of providing transactional support for fast, non-volatile memories that exploits their raw performance and makes programming easy.

First, we present a prototype PCIe-based storage array that targets fast, non-volatile memories and provides hardware support for multi-part atomic write operations. Multi-part atomic writes atomically and durably commit groups of writes to storage. Unlike previous approaches for flash-based SSDs, multi-part atomic write support makes logging scalable and transparent, providing a strong foundation for flexible ACID transactions. Using multi-part atomic writes, existing transactions mechanisms such as ARIES-style write-ahead logging can be redesigned to make optimal use of these memories, providing up to 3.7× the performance of the baseline version of ARIES.

Second, we address the problem of providing strong consistency guarantees for storage that is directly accessible via the processor's memory bus. We present NV-heaps, a persistent object store which provides a familiar programming interface and protects against application and system failures by avoiding familiar programmer errors as well as new errors that only arise with persistent objects. Compared to Berkeley DB and Stasis, two persistent object stores designed for disk, NV-heaps improves performance by 32× and 244×, respectively, for operations on a variety of persistent data structures. To further improve safety, we present programming language support for NV-heaps. We introduce a Java-like language that provides the features NV-heaps require, along with a new static dependent type system that enforces the invariants that make NV-heaps safe.

Chapter 1

Introduction

Emerging non-volatile memory technologies such as phase change memory, spin-torque transfer memory, and the memristor will revolutionize the role of storage in computing. The introduction of these technologies represents an orders of magnitude decrease in latency along with dramatic increases in bandwidth relative to hard disks and flash memory. As these technologies become available via fast interconnects close to the processor, the performance of storage will approach or equal the performance of main memory. This fundamental shift in the balance of storage, system bus, main memory, and CPU performance challenges the traditional assumptions behind modern computer systems.

With disk as the de facto standard storage technology, decades of work in software is predicated on the enormous gap between memory and storage performance. But this gap will shrink or disappear entirely with the arrival of fast, non-volatile memories. As a result, the current overheads required to access non-volatile storage (i.e., microseconds for IO system calls) will severely limit performance. The performance penalties of the operating system and file system will void many of the performance gains provided by these new memory technologies, so removing these overheads will be crucial to realizing their full potential.

In addition, new non-volatile memory technologies will offer a large increase in flexibility compared to disks, particularly in their ability to perform fast, random accesses. Unlike flash memory, these new technologies will support in-place updates, avoiding the extra overhead of a translation layer. Further, these new memories can present a DRAM-like interface to storage, unlike disks and flash which require a block-based interface. This removes the sector restriction on IO read and write patterns, which means that applications may no longer need to package their data into long byte streams for efficient transfer to and from storage.

Improved performance and flexibility will redefine the notion of non-volatile data in applications. Currently, our applications and the tools we use to access storage assume disk is the backing store. Hence, non-volatile data is treated as something that must be accessed in large, sequential blocks whenever possible. Consequently, we rely on untyped, heavy-weight file abstractions to access storage. However, with fast, non-volatile memories, we can potentially manipulate non-volatile data in the same way we manipulate volatile data: We use language type constructs (structs and classes) combined with light-weight load and store instructions.

This thesis is based on a model for storage that exposes non-volatile memories directly to the user with minimal software overheads. In the common case, the operating system is removed from the critical path to exploit the full performance of the storage technology. With a fast and flexible interface, the user can adapt an application to best match its requirements with the characteristics of the storage device.

In many cases, we have to maintain compatibility with legacy software and we need to minimize the programming effort to port our applications to a new storage device. Treating fast, non-volatile memories as a block device achieves this goal, as it requires little to no changes to existing code. The application can immediately run on the storage array. However, performance gains tend to be limited because existing software is usually optimized for disk. These optimizations tend to focus on ways to cache data in DRAM, and they do not often take advantage of the performance of fast, non-volatile memories.

Alternatively, we may utilize the flexible interface of fast, non-volatile memories to build persistent data structures directly in storage. Instead of reading bytes serially from a file and building data structures in memory, the data structures would appear, ready to use in the program's address space, allowing quick access to even the largest, most complex persistent data structures. This leverages decades of work in data structure design and integrates well with existing programming languages, giving the programmer considerably more power over traditional, untyped file IO. However, it does require significant changes to existing code.

Whether we continue to access storage as a block device or access it directly like memory, we need guarantees that our data will not be corrupted if there are failures. Strong consistency guarantees are what ultimately make storage useful. Applications such as file systems, databases, and persistent object stores all depend on being to move from one consistent state to the next. When storage is exposed directly to the user, it is even more challenging to provide these guarantees. One wrong pointer assignment or a system failure can permanently corrupt a persistent data structure. The narrow, block-based interface of disk actually provides some protection against corruption, whereas making storage accessible via loads and stores does not. Data can be corrupted in several ways: programmer errors, stray writes on the memory bus, or system failures such as an application/OS crash or power loss.

This dissertation focuses on system support for transactions in next-generation, non-volatile memories. The goal of our work is to make accessing storage as fast as accessing the underlying technology directly, to make programming easy, and to provide strong consistency guarantees in the face of failures. These three criteria, when taken together, are challenging to meet. For example, transactions require some form of logging which, at a minimum, doubles the amount of data written to the device. Transactions can also require complex management of the log space and this should be hidden from the user. By providing system support, both in software and hardware, we can keep these overheads low and provide strong safety guarantees.

To understand the requirements for system support, we examine technology trends in non-volatile memories and storage devices. Chapter 2 explains these trends and presents two prototype storage system architectures based on fast, non-volatile memories. The first architecture is an advanced storage array available over PCIe interconnect which provides a flexible interface to access data of arbitrary size and alignment. IO requests are serviced with a user-space driver that minimizes software overheads by bypassing the operating system and file system in the common case. The second architecture attaches storage directly to the processor's memory bus. We memory map a region of the physical address space corresponding to storage, making it accessible with load and store instructions. Both architectures exploit the performance of the underlying storage technology, but, by themselves, do not provide any guarantees against failures.

The overheads of providing strong consistency guarantees are high. Transactions require locking, logging, and recovery implementations to ensure data integrity in the face of failures. Existing systems that provide powerful transaction mechanisms typically rely on write-ahead logging (WAL) implementations that were designed with slow, disk-based storage systems in mind. However, emerging, non-volatile memory technologies present performance characteristics very different from both disks and flash-based SSDs, forcing us to reexamine how best to support transactions.

Chapter 3 addresses the problem of implementing application-level transactions in fast, non-volatile memory-based storage systems. We examine the features that a system like ARIES [MHL$^+$92], a WAL algorithm popular for databases, must provide and separate them from the architectural decisions ARIES makes to optimize for disk-based systems. We present a new WAL scheme optimized for non-volatile memories, called MARS, in tandem with a novel SSD multi-part atomic write primitive that combine to provide the same features as ARIES without any of the disk-centric baggage. The new atomic write primitive makes the log's contents visible to the application, allowing for a simpler and faster implementation. MARS provides atomicity, durability, and high performance by leveraging the enormous internal bandwidth and high degree of parallelism that advanced SSDs will provide. We present an implementation of MARS and the our novel atomic write primitive in a prototype next-generation SSD. We demonstrate that the overhead of the primitive is minimal compared to normal writes, and our hardware provides large speedups for transactional updates to hash tables, b-trees, and large graphs. Finally, we show that MARS outperforms ARIES by up to 3.7× while reducing software complexity.

In addition to adapting existing systems for emerging memory technologies, we also explore a new abstraction for fast storage. Chapter 4 presents NV-heaps, a system designed to provide fast and safe access to persistent data through an intuitive and familiar programming model. NV-heaps is a persistent object store that targets storage attached to the processor memory bus. NV-heaps protects against application and system failures by avoiding familiar bugs such as dangling pointers, multiple frees, and locking errors. It also prevents new types of hard-to-find pointer safety bugs that only arise with persistent objects. These bugs are especially dangerous since any corruption they cause will be permanent. NV-heaps provides the following features for building persistent data structures: persistent objects, specialized pointers, memory management, and atomic sections. We describe the implementation of NV-heaps and how it achieves ACID semantics. We implement a variety of persistent data structures using NV-heaps, BerkeleyDB [SO92], and Stasis [SB06]. Our results show that NV-heaps outperforms BerkeleyDB and Stasis implementations by 32× and 244×, respectively, by avoiding the operating system and minimizing other software overheads.

While NV-heaps present a new abstraction for storage that provides strong safety guarantees, there are several potential issues that must be addressed. First, there are performance overheads due to the features they provide, and it may not make sense for every application to pay for all these features when it does not need them. Second, the flexibility of directly accessing storage attached to the memory bus creates opportunities for data corruption that do not exist in a system with a well-defined and restricted interface (e.g. block-based file IO in a database). Third, there are limitations to what NV-heaps can guard against as a result of our library-based implementation. For example, programmers can perform arbitrary pointer arithmetic, circumventing our smart pointer types that normally guarantee safe operations on references. In Chapter 5, we present programming language support and a series of programming models that address these issues. We introduce a core language based on Java that contains the features that NV-heaps require, and we describe a novel static dependent type system that enforces the necessary invariants about NV-heaps and references.

Finally, in Chapter 6 we summarize the contributions of this dissertation, including the MARS architecture and atomic write support, the design of NV-heaps, and language support for persistence.

Acknowledgments

This chapter contains material from "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories", by Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which appears in *ASPLOS '11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2011 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

This chapter contains material from "From ARIES to MARS: Reengineering Transaction Management for Next-Generation, Non-Volatile Memories", by Joel Coburn, Trevor Bunker, Rajesh K. Gupta, and Steven Swanson, which will be submitted for publication. The dissertation author was the primary investigator and author of this paper.

This chapter contains material from "Programming Language Support for Fast, Byte-Addressable, Non-Volatile Memory", by Joel Coburn, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which will be submitted for publication. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Storage Trends

In this chapter, we examine recent trends in non-volatile memory technologies and the corresponding trends in storage devices, highlighting the enormous performance gains that are possible. We then identify the overheads in the existing IO software stack that ultimately limit the performance of these new storage devices to a level far below their capability. To overcome these problems, we present prototypes for two possible storage architectures: (1) a storage array of non-volatile memories accessible to the system through PCIe interconnect, and (2) non-volatile memories in DIMMs attached to the processor's memory bus in a manner similar to DRAM. These prototypes serve as the experimental test beds for the rest of the work in this thesis. We briefly discuss the architecture and interface of these prototypes, and how each design removes software overheads to make access to storage fast and flexible.

2.1 Storage Trends

Emerging non-volatile memory technologies will present a DRAM-like interface and achieve performance that is within a small factor of DRAM in both latency and bandwidth. These technologies are vying to replace flash memory as the dominant storage technology in solid-state drives. They also have the potential to replace DRAM as main memory as classic CMOS scaling begins to falter.

Table 2.1 describes several of the most promising technologies, using flash

Table 2.1: **Memory technology summary** Phase change and spin-torque transfer memories provide performance near that of DRAM while maintaining their data in the absence of power.

Technology	Latency		Endurance
	Read	Write	
DRAM	25ns	35ns	10^{18}
Spin-Torque Transfer [TKM$^+$07]	29ns	95ns	10^{15}
Phase Change Memory [BRK$^+$04, LIMB09]	48ns	150ns	10^{8}
Flash Memory SLC	25us	200us	10^{5}

memory and DRAM as reference points. Flash memory, although much faster than disk especially for random accesses, has very asymmetric performance as shown by the large difference in read and write latencies. This is due to the program-erase nature of flash. As process geometries shrink and manufacturers push for higher densities, the reliability of flash gets worse: The latest multi-level cell (MLC) technologies become unusable after several thousand program-erase cycles. In contrast, DRAM has nearly infinite write endurance and read and write latencies tend to be nearly equal. However, data stored in DRAM does not persist beyond system failures or power cycles.

Phase-change memory (PCM), one of the most mature technologies, stores data as the crystalline state of a chalcogenide metal layer [Bre08]. PCM may eventually surpass flash memory in density according to the ITRS [ITR09], and recent work has demonstrated that it has the potential to become a viable main memory technology [LIMB09, QSR09, ZZYZ09]. While it has a higher endurance than flash memory, PCM does pose some reliability concerns: A PCM bit has a typical lifetime of 10^8 write cycles after which point the bit can no longer be programmed. To mitigate this problem, writes should be spread out across the different bits in the device, maximizing the time to bit failure. Several PCM wear leveling schemes are already available to address this issue [DAR09, LIMB09, ZZYZ09, CL09, QKF$^+$09a].

Spin-torque transfer memory (STTM) stores bits as a magnetic orientation of one plane in a magnetic tunnel junction (MTJ). Depending on the orientation, the junction's resistance is either low (the "anti-parallel" state) or high (the "parallel" state) [DSPE08]. In this respect, STTM is similar to previous magnetic RAM technologies. STTM differs in how it sets the orientation in the MTJ: Instead of using electric fields as previous MRAM technologies have, STTM uses a current of polarized electrons. This avoids the scaling limitations that plagued field-based devices. In the near future, STTM's density, latency, and power consumption may approach those of DRAM. STTM does not suffer from the endurance problems of PCM or flash memory.

Unlike flash, these new non-volatile memory technologies do not require a separate erase operation to clear data before a write. This makes in-place updates possible and, therefore, eliminates the complicated flash translation layer that manages a map between logical storage addresses and physical flash storage locations to provide the illusion of in-place updates. PCM still requires wear-leveling and error correction, but there are several efficient solutions to both of these problems [QKF$^+$09b, QSLF11, SLSB10, ICN$^+$10, YMC$^+$11]. With fast, in-place updates, the end-to-end performance of a non-volatile memory storage array can be very close to the performance of the underlying memory technology.

As the performance of storage technologies continues to evolve, so will the architecture of storage devices themselves. Unlike traditional spinning disks which must perform requests serially, storage devices composed of non-volatile memories possess an enormous amount of internal parallelism and bandwidth, making it possible to process many outstanding requests at once. This requires an advanced storage controller that can buffer and manage a large amount of state. Further, the improvements in latency and bandwidth offered by non-volatile memories can only be exploited with corresponding improvements in the rest of the system to make access to data fast. This means storage devices will use high performance interconnects and move closer in proximity to the processor.

Table 2.2 shows the latency and bandwidth for performing a 4 KB read from user space for current and future storage devices. A RAID array of four disks serves as the reference point: It takes 7.1 ms to complete the request at 2.6 MB/s. In 2007, PCIe-based flash SSDs were introduced by Fusion-IO [fus], and these devices provide around 100× improvement in latency and bandwidth over disks. In the near future, we envision PCIe-based SSDs containing an array

Table 2.2: **Performance for a 4 KB read** Latency and bandwidth improve at rates of 2.5× and 2.6× per year, respectively, as storage technologies and interconnect improve over time.

Storage Device	RAID-Disk	PCIe-Flash (2007)	PCIe-NVM (2013)	DDR-NVM (2016)
Latency (μs)	7,100 1×	68 104×	8.2 865×	1.5 4,733×
Bandwidth (MB/s)	2.6 1×	250 96×	1,600 669×	14,000 5,384×

of PCM. Using a recent prototype (to be discussed in Section 2.3), a single read request takes just 8.2 μs and achieves a bandwidth of 1.6 GB/s. When PCM is put on DIMMs and placed on the processor's memory bus alongside DRAM, access latency shrinks to 1.5 μs and bandwidth is 14 GB/s. This results in a 4,733× latency improvement and 5,384× improvement in bandwidth. If storage devices follow the performance numbers presented here, then latency improves at a rate of 2.5× per year and bandwidth improves at a rate of 2.6× per year. Both of these rates outpace Moore's Law, but realizing these performance targets will not be possible with existing system architectures.

2.2 Overhead of the Software Stack

The performance of storage devices based on technologies such as PCM and STTM in modern computing systems will be severely limited by our existing software infrastructure, and this is already evident in high-performance flash SSDs. The traditional software stack for storage was designed for disk and requires all requests go through the operating system and the file system using system calls such as `read` and `write`.

Figure 2.1 shows the overhead of software, broken down into components from the OS, the file system, and the hardware, for a 4 KB read request from user space. For disk, the hardware latency is over 390× the latency of the OS and file system combined, making software overheads negligible. For a flash-based SSD, the hardware latency drops to about 13× the latency of software. For a PCIe-based SSD, the distribution of latencies shifts: the OS and the file system take 2.3× as

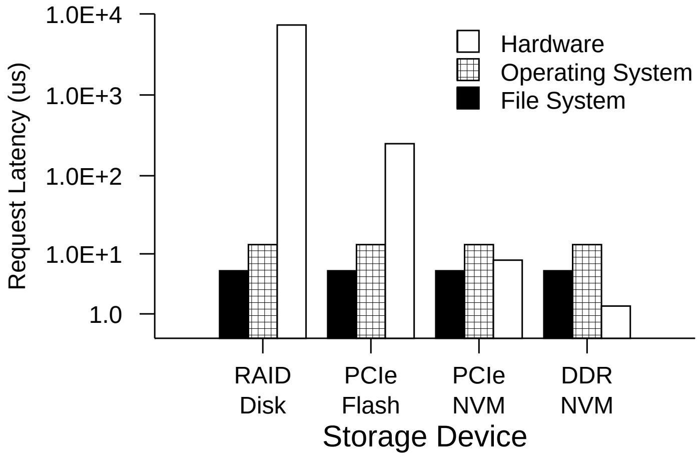


Figure 2.1: **Latency breakdown for a 4 KB read** The latency for a read request is decomposed into three parts: the operating system overhead, the file system overhead, and the hardware latency. For fast, non-volatile memories, the software overhead makes up the majority of the total access time.

long as the hardware to service the request. For PCM attached to the processor's memory bus, the situation is even worse: software is 12.4× slower than hardware. Using the existing software stack to access storage destroys the performance gains of fast, non-volatile memories. In the next section, we describe two prototype storage architectures that lower or eliminate these software overheads.

2.3 New Storage Architectures

In Figure 2.2, we show a computer system with two storage devices using fast, non-volatile memories. The first is a prototype storage array called Moneta [CDC^{+}10] which sits on the PCIe bus and houses a large array of storage. The second is a collection of non-volatile memories in DIMMs attached to the processor's memory bus. Because non-volatile memories are still in development and have not yet matured to their performance targets, we use DRAM to emulate them. In the following subsections, we describe each storage device, its interface, and how we lower or eliminate software overheads.

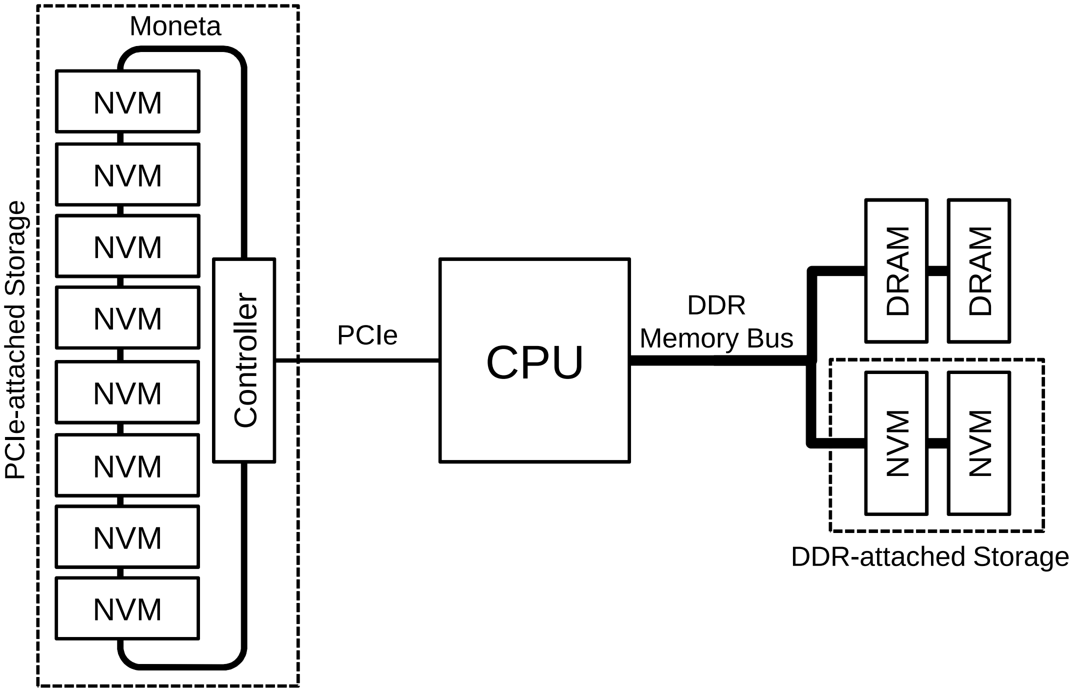


Figure 2.2: **System diagram with prototype storage devices** Fast, non-volatile memories will appear in PCIe-attached storage arrays and in DIMMs connected to the processor's memory bus.

2.3.1 PCIe-attached Storage Array

Moneta [CDC$^+$10] is a PCIe-based SSD designed around non-volatile memories like PCM. It spreads 64 GB of storage across eight memory controllers connected via a high-bandwidth ring. Each memory controller provides 4 GB/s of bandwidth for a total internal bandwidth of 32 GB/s. An 8-lane PCIe 1.1 interface provides a 2 GB/s full-duplex connection (4 GB/s total) to the host system. The prototype runs at 250 MHz on a BEE3 FPGA prototyping system [bee].

The Moneta storage array emulates advanced non-volatile memories using DRAM and modified memory controllers that insert delays to model longer read and write times. We model PCM in this work and use the latencies from [LIMB09] (48 ns and 150 ns for array reads and writes, respectively). Moneta uses start-gap wear leveling implemented at the memory controllers [QKF$^+$09b].

Moneta is accessible through the Linux IO stack and uses a customized device driver. The design relies on hardware and software optimizations such as bypassing the Linux IO scheduler, removing unnecessary context switches, and removing locks in the driver in order to reduce latency and maximize concurrency. Compared to the baseline IO stack, these changes reduce latency by 62%. However, the remaining software overhead is still quite high, with the system call and file system overheads accounting for 65% of the latency.

Recent work [CME$^+$12] provides a user-space driver that eliminates much of remaining software overhead by bypassing the OS and file system for most accesses. The user-space driver provides a private, virtualized interface for each process and offloads file system protection check into hardware. Each application communicates directly with the storage array via a private set of control registers, a private DMA buffer, and a private set of 64 tags that identify in-flight operations. To enforce file protection, the user space driver works with the kernel and the file system to download extent and permission data into Moneta, which then checks that each access is legal. Consequently, accesses to file data do not involve the kernel at all in the common case. However, modifications to file system metadata still go through the kernel. Applications can use the new interface without modification, since the library interposes on file access calls. With the user-space interface, Moneta performs 4 KB IO operations up to 60% faster than going through the kernel and throughput increases by 7.6×.

2.3.2 DDR-attached Storage

In the near future, we will see systems with high-performance non-volatile memory on the memory bus capable of providing a storage capacity ranging from gigabytes up to terabytes. However, mature products based on these memories will take several years to appear. We prototype such a storage configuration by emulating technologies such as PCM using existing DRAM DIMM modules. We assume that bus negotiation and transfer times are similar to existing DRAM interfaces, and the only additional overhead for non-volatile accesses arise from the difference in memory technology.

We consider two methods of accessing storage attached to the memory bus:

direct access using load and store instructions, and access through a traditional block-based interface using `read()` and `write()` system calls. We present an emulation system for each method. Both of these systems run applications for many billions of instructions while simulating the performance impact of using advanced, non-volatile memories.

Modeling byte-addressable storage

Storage attached to the memory bus can be made accessible without going through a block-based interface because it is directly exposed to the processor in a manner similar to DRAM. The region of physical address space corresponding to non-volatile storage can be memory mapped into an application's virtual address space. This makes the data accessible via normal load and store instructions. Normally, the kernel copies memory mapped data between a block device and DRAM, but in this case, copying is not necessary.

Our first emulation system models the latency for memory-level load and store operations to fast, non-volatile memories on the processor's memory bus. The system uses Pin [LCM$^+$05] to perform a detailed simulation of the system's memory hierarchy augmented with non-volatile memory technology and epoch barriers [CNF$^+$09], which are an architectural feature we use to guarantee an ordering of updates to memory (described in more detail in Section 4.3.2). The memory hierarchy simulator accounts for both the increased array read time and the added delay between a write and the operations that follow, allowing it to accurately model the longer read and write times for PCM and STTM memories. For PCM we use the performance model from [LIMB09] which gives a PCM read time of 67 ns and a write time of 215 ns. We model STTM performance (29 ns reads and 95 ns writes) based on [TKM$^+$07] and discussions with industry. The baseline DRAM latency for our system is 25 ns for reads and 35 ns for writes, according to the datasheet.

Our emulation system divides program execution into intervals of one billion instructions. At the beginning of each interval, we turn on instrumentation to perform a detailed cache simulation of the first 100 million instructions. The simulation provides the average latency for last level cache hits and misses and for the epoch barriers. After the simulation phase, we use hardware performance counters to track these events on a per-thread basis, and combine these counts with the average latencies to compute the total application run-time. The model assumes that memory accesses execute serially, which makes our execution time estimates conservative.

There are several potential problems with this system. First, program behavior may change during the interval, invalidating the execution signature we collect during simulation. To mediate this problem, we annotate the applications with a special function call between phases that triggers the start of a new interval. Our applications have predictable, consistent behavior, so identifying phase boundaries is easy. For more complex workloads, a phase-based sampling methodology such as [SPC01] could be used. Second, this methodology does not capture fine-grain parallelism among accesses to non-volatile memory. This is a conservative assumption, since it assumes that all non-volatile memory accesses within a thread occur sequentially.

To calibrate our system we used a simple program that empirically determines the last-level cache miss latency. We ran the program with the simulated PCM and STTM arrays and its estimates matched our target latencies to within 10%.

The overhead due to Pin's instrumentation varies unpredictably with thread count and application, although it is consistent for a particular application and thread count combination. To account for this overhead, we run the instrumented version several times and compare it to the run-time without Pin. We then subtract this value from the Pin instrumented run-times we report. This methodology delivers results accurate to within 5%.

Modeling a block device based on fast, non-volatile memory

Our second emulation system presents a block device interface similar to the one available for disk. This makes storage accessible using `read()` and `write()` system calls, which require going through the OS, file system, and device driver.

To model a non-volatile memory-based block device, we modified the Linux RAM disk driver to let us insert extra delay on accesses to match the latency of non-volatile memories. Measurements with a simple disk latency benchmark show that the emulation is accurate to within about 2%. This emulation system provides us with a baseline in our experiments that demonstrates the overheads of the existing IO software stack.

2.4 Summary

The two storage architectures just presented remove software overheads to expose the performance of the underlying non-volatile memory technology. While this dramatically improves performance for raw read and write operations, it does not necessarily translate into commensurate application-level gains, as shown in a recent study [CCM$^+$10]. The primary reason for this is that applications are often highly optimized for disk, and these optimizations tend not to take full advantage of fast, non-volatile memories. In particular, the way in which databases and other applications provide consistency guarantees for their data tends to be very disk-specific. In the next chapter, we address this problem by examining existing transaction mechanisms that rely on write-ahead logging, and we propose a new approach designed for fast, non-volatile memories.

Acknowledgments

This chapter contains material from "Providing Safe, User Space Access to Fast, Solid State Disks", by Adrian M. Caulfield, Todor I. Mollov, Louis Eisner, Arup De, Joel Coburn, and Steven Swanson, which appears in *ASPLOS '12: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems.* The dissertation author was the fifth investigator and author of this paper. The material in this chapter is copyright ©2012 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use

is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

This chapter contains material from "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories", by Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which appears in *ASPLOS '11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems.* The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2011 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

This chapter contains material from "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories", by Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson, wwhich appears in *MICRO-43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* The dissertation author was the third investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for

Chapter 3

Redesigning Transaction Mechanisms for Advanced SSDs

As discussed in the previous chapter, emerging fast non-volatile memory (NVM) technologies are orders of magnitude faster than existing storage technologies (i.e., disks and flash). This increase in performance shifts the balance between storage, system bus, main memory, and CPU performance and will force designers to reorganize storage architectures to maximize application gains and exploit memory performance and parallelism. While recent work focuses on optimizing read and write performance for storage arrays based on these memories [CDC$^+$10, CME$^+$12], systems must also provide strong guarantees about data integrity in the face of failures.

Applications such as file systems, databases, persistent object stores, and other persistent data structures are only useful if they provide strong consistency guarantees. Typically, these applications use some form of transaction to move the data from one consistent state to another. Most systems implement transactions using software techniques such as write-ahead logging (WAL) or shadow paging. These techniques are based on complex, disk-based optimizations designed to minimize the cost of synchronous writes and leverage the sequential bandwidth of disk.

NVM technologies provide very different performance characteristics, and exploiting them requires new approaches to implementing application-level transactional guarantees. NVM storage arrays provide parallelism within individual chips, between chips attached to a memory controller, and across memory controllers. In addition, the aggregate bandwidth across the memory controllers in an NVM storage array will outstrip the interconnect (e.g., PCIe) that connects it to the host system.

In this chapter, we present a novel WAL scheme, called MARS, optimized for NVM-based storage. The design of MARS is based on an examination of ARIES [MHL$^+$92], a popular WAL-based recovery algorithm for databases, that separates the features it must provide from the architectural decisions it is built on that optimize for disk-based systems. MARS uses a multi-part atomic write primitive to implement ACID transactions on top of a novel NVM-based SSD architecture. As we will show, multi-part atomic writes are a useful building block for a range of applications and transaction mechanisms in addition to ARIES.

The multi-part atomic write interface supports atomic writes to multiple portions of the storage array without alignment or size restrictions, and the hardware shoulders the burden for logging and copying data to enforce atomicity. This interface exposes the logs to the user and allows the user to manage the log space directly, providing greater flexibility for software to implement transactions. In contrast, recent work on atomic write support for flash-based SSDs [PRZ08, ONW$^+$11] hides the logging in the flash translation layer, restricting user interaction with the logs.

We will present an implementation of multi-part atomic writes in the Moneta [CDC$^+$10] PCIe-based storage array. Our design achieves high performance by distributing logging, commit, abort, and recovery functions across multiple memory controllers to leverage the internal bandwidth of the storage device. Shifting support for logging and commit into hardware relieves pressure on the PCIe link and minimizes operating system overhead, since issuing an atomic write requires just a single IO request and a single DMA transfer.

The remainder of this chapter is organized as follows. Section 3.1 examines existing transaction mechanisms in the context of fast NVM-based storage, deconstructs ARIES, and proposes MARS as an alternative for fast NVM-based storage. In Section 3.2, we describe a new set of IO primitives that support multi-part atomic write operations for MARS and other applications. Section 3.3 places this work in the context of prior work on support for transactional storage. In Section 3.4, we describe the hardware architecture in detail. Section 3.5 evaluates our multi-part atomic write support and its impact on the performance of MARS and other persistent data structures. Section 3.6 summarizes the contributions of MARS and multi-part atomic writes.

3.1 Revisiting Transaction Support

This section examines existing transaction mechanisms, focusing on ARIES write-ahead logging. We describe the features that a system like ARIES must provide to implement flexible ACID transactions. We analyze the design decisions that make ARIES a good fit for disk but are not well-suited for fast NVM-based storage. Then, we propose MARS, a novel WAL architecture that takes advantage of the multi-part atomic writes provided by our prototype storage array. MARS provides the features required by ARIES while exploiting the performance of fast NVMs.

3.1.1 Transaction mechanisms

Transaction implementations depend on the requirements of the application and the underlying storage technology. For many applications, relational databases are a good fit because they provide full ACID semantics and accommodate a wide variety of data formats and operations on that data. Many databases are built on top of ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) [MHL$^+$92], a powerful algorithm for providing strong consistency guarantees. ARIES-style transactions are scalable and support different levels of isolation.

For web services or file systems, simpler approaches are often the best option because the class of transactions they must support is narrower. Transaction size is often fixed or bounded, and transactions often need not have the flexibility to read back or update data multiple times. Previous systems [GHKdJ96, PRZ08, ONW$^+$11] provided an atomic write interface that is limited to these types of transactions. The writes would be batched up in memory and sent to the storage array in a single IO operation. Our multi-part atomic write interface, on the other hand, allows transactions to be specified in multiple IO requests, offering scalability and better programmability. Also, this interface provides visibility to each logged part of a transaction prior to commit.

Transaction implementations typically include a concurrency control scheme (e.g. two-phase locking [BHG87]), some form of data versioning, and a recovery algorithm. The most common ways to implement data versioning are either by using write-ahead logging and updating data in-place or by using shadow paging. Both of these techniques are optimized heavily for disk. With this is mind, we now re-examine existing transaction mechanisms in the context of fast NVM-based storage and the high-level features that applications demand.

3.1.2 Deconstructing ARIES

We focus on ARIES because it influenced the design of many industrial-strength databases and is a key building block in providing fast, flexible, and efficient ACID transactions. ARIES uses WAL and has been tuned to exploit the sequential write performance of disk. In Table 3.1, we list several of the important *features* that ARIES provides to higher-level software (e.g., the rest of the database) and make it useful to a variety of applications. For example, ARIES offers flexible storage management since it supports objects of varying length. It also allows transactions to scale with the amount of free disk storage space rather than with available main memory. Features like operation logging and fine-grained locking improve concurrency. Recovery independence makes it possible to recover some portion of the database even when there are errors. Independent of the underlying storage technology, ARIES must export these features to the rest of the database.

To provide these features and achieve high performance, ARIES incorporates a set of *design decisions* (Table 3.2) that exploit the properties of disk: They optimize for long, sequential accesses and avoid short, random accesses whenever

Table 3.1: **ARIES features** ARIES-style WAL provides the above features to the rest of the system regardless of storage technology.

Feature	Benefits	Available in MARS?
Flexible storage management	Supports varying length data	Yes
Fine-grained locking	High concurrency	Yes
Partial rollbacks via savepoints	Robust and efficient transactions	Yes
Operation logging	High concurrency lock modes	N/A
Recovery independence	Simple and robust recovery	N/A

possible. Also, because disk drives are effectively serial devices, these design decisions do not exploit parallelism in the backing store. As we will show, this makes them a poor fit for advanced, solid-state storage arrays which provide fast random access, high internal bandwidth, and a high degree of parallelism. Alternatively, we present a novel multi-part atomic write IO primitive which can exploit the characteristics of fast NVM-based storage. In Section 3.1.3, we describe the design of MARS, which is a new version of ARIES engineered to take advantage of multi-part atomic writes. Next, we discuss each of the major design decisions behind ARIES.

No-force In ARIES, the system writes log entries to the log in storage before any changes to an object are written to storage. Then, if a crash occurs, ARIES can redo the partially completed operation. To hide the latency of random writes to disk, ARIES implements a *no-force* policy, which means the system writes updated pages back to disk after commit. ARIES flushes redo log entries to disk in a synchronous sequential write during commit, making the updates available for the recovery routine to reapply in case of a failure. In fast NVM-based storage, however, random writes are no more expensive than sequential writes, so the value of a no-force policy is much lower.

Steal ARIES relies on a steal policy which improves performance for disk but provides little to no benefit for fast NVM-based storage. A *steal* policy allows the buffer manager to write dirty pages back to disk before commit. By stealing pages for early write back, the buffer manager can reclaim buffer space during transaction

Table 3.2: **ARIES design decisions** ARIES relies on a set disk-centric optimizations to maximize performance on conventional storage systems. However, these optimizations are a poor fit for the characteristics of storage based on fast, non-volatile memories. Instead, MARS uses an alternative set of design decisions.

Design decision	Advantage for disk	Implementation
No-force	Eliminate synchronous random writes	ARIES: Flush redo log entries to storage on commit
		MARS alternative: Force write backs at memory controllers
Steal	Reclaim buffer space Eliminate random writes Avoid false conflicts	ARIES: Write undo log entries before writing back dirty pages
		MARS alternative: Hardware does all in-place updates and the log always holds the latest copy of the data
Pages	Simplify recovery and buffer management	ARIES: Perform updates on pages assuming page writes are atomic
		MARS Alternative: Hardware uses pages and software operates on objects
Log Sequence Numbers (LSNs)	Simplify recovery Enable high-level features	ARIES: Order updates to storage using LSNs
		MARS Alternative: Hardware enforces ordering with commit sequence numbers

execution (supporting larger transactions), group writes together to take advantage of sequential disk bandwidth, and avoid data races on pages shared by overlapping transactions. Stealing requires undo logging because it is only safe to write back dirty pages if copies of old values have been written to disk. After a crash or abort, the system may use the undo log entries to recreate the overwritten data.

For disk, the performance benefits greatly outweigh the overhead of the extra logging. With fast NVMs, because the performance of random writes and sequential writes is the same, the overhead of undo logging can actually hurt overall performance. The cost of writing a log entry to storage before making an update occurs on every update, but the benefit of writing pages back early occurs far less frequently. While stealing eliminates costly seek time for disk, writing pages back early as part of a larger write to fast NVM-based storage only helps amortize the setup/completion cost of an IO request.

Pages and LSNs In ARIES, disk pages are the basic unit of recovery and each page contains a log sequence number (LSN). LSNs provide an ordering on disk updates. At recovery, ARIES uses LSNs to decide which updates to reapply to bring the system into a consistent state. While the design of ARIES is not restricted to pages per se, pages simplify the implementation of recovery. Assuming a single page write is atomic, the system uses them as a foundation for larger atomic updates. When the system logs an update, it writes the LSN in the same page as its matching log record, guaranteeing that the two are updated atomically. To be useful for recovery, LSNs must be generated with a unique order and must be written out to disk in that order [JPS$^+$10]. This adversely affects performance. It also complicates situations where objects span multiple pages or multiple objects fit in a single page. Recent work [SB09] proposes segments as an alternative to pages, making it possible to efficiently handle objects of various sizes and copy them directly between the application and storage array.

Pages and LSNs are even more restrictive for fast NVM-based storage arrays because they limit parallelism, waste bandwidth, and increase latency. Maintaining headers in log records and forcing log records out to disk in LSN order serializes execution, resulting in under-utilization of the storage array. Because objects may share pages, LSNs may artificially order updates when the system could in fact perform those updates in parallel. Also, when objects consume less than a page of storage, the system must pay the additional cost in IO processing to update an entire page. This is particularly wasteful because fast NVM-based storage has no sector or page restriction on access size and can handle arbitrarily-sized requests efficiently.

3.1.3 Building MARS

We now describe the design of MARS, an alternative transaction mechanism based on ARIES but adapted to the characteristics of fast NVM-based storage. MARS relies on our multi-part atomic write primitive, presented in Section 3.2, and ensures that the most recent copy of an object is always directly accessible, whether the most recent copy lives at the object's home location or somewhere in a log. Using multi-part atomic writes, MARS can eliminate the need for pages and LSNs. MARS replaces the no-force and steal policies designed for disk with more efficient mechanisms that utilize the internal bandwidth of the storage array and the flexible interface of our IO primitive. For each design option in Table 3.2, we propose an alternative method better suited to fast NVM-based storage.

No-force Instead of performing in-place updates asynchronously from software, we implement a *force* policy in hardware at the memory controllers. This takes advantage of Moneta's large internal bandwidth—32 GB/s at the memory controllers compared to 4 GB/s PCIe link bandwidth—and eliminates the extra IO requests required for commits and write backs. Also, moving write backs into hardware has other benefits: It eliminates the need for checkpointing the log, and the system immediately reclaims the log space. We choose a force policy over no-force because it allows our hardware to utilize idle cycles, make better use of limited hardware transaction resources, and minimize the amount of work needed to be done at recovery time.

Steal Instead of writing dirty pages back early, we propose simply dropping pages from the buffer pool to acquire free space when needed. Our multi-part atomic write architecture makes this possible: The system first writes an update out to the log and then proceeds to update the buffer pool page. Unlike other systems [GHKdJ96, PRZ08, ONW$^+$11], we do not wait to flush the log at commit. Consequently, the system can page in the updates from the log later as needed. To do this, the system must maintain a mapping of buffer pool pages to log entries, which is possible using our atomic write interface because software controls the placement of log entries in the log files (see Section 3.2).

Pages and LSNs Because fast NVM-based storage directly supports updates of arbitrary sizes and our IO primitive makes those updates atomic, MARS can eliminate the use of pages as the basic unit of update. Objects are visible to software in a contiguous and unmodified form while our hardware support keeps track of objects internally using pages. This means that MARS pays for only the amount of storage it needs. It also means that it is possible to maintain the same contiguous layout of application-level objects in both storage and main memory. This has two chief advantages. First, it avoids the significant cost of translating objects back and forth between pages and their native contiguous format. Second, because software no longer needs to intervene on a per-page basis, it enables the use of DMA and zero-copy IO operations [SB09].

Our multi-part atomic write interface eliminates the need for software managed and enforced LSNs. Instead, the storage array maintains ordering in hardware by assigning a unique commit sequence number to a transaction at commit time. This effectively removes the serialization of write requests due to LSNs, allowing log writes from different transactions to proceed in parallel.

With an implementation based on multi-part atomic writes, MARS provides many of the features (shown in Table 3.1) that ARIES exports to higher-level software while significantly reducing software complexity. MARS provides flexible storage management and fine-grained locking by making objects directly accessible. Partial rollbacks are achieved using an abort function provided by our hardware that can rewind to any point in the log. Operational logging and recovery independence are currently out of the scope of our atomic write primitive, requiring customizations to the interface specific to ARIES. However, they are a possible topic for future work.

3.2 Multi-part atomic write overview

To take advantage of storage array architectures based on fast NVMs, we present a novel multi-part atomic write interface that provides efficient and safe updates to storage. Multi-part atomic writes offer a simple, flexible, and general-purpose way to implement transactions at the application-level. This section describes our atomic write interface, highlighting how transactions execute and how the interface makes the log visible to the application. We discuss the rationale behind our design.

3.2.1 The transaction model and interface

Our system provides the means to group multiple write operations into transactions and ensure they execute atomically and durably. To achieve full ACID semantics, the application implements consistency and isolation in software. The writes in a transaction can be scattered throughout the storage array and be of any size or alignment. The total size of data that a transaction can update is limited only by the space available for storing the log in the storage array.

Applications create and execute transactions using the commands in Table 3.3. Each application accessing the storage device has a private set of 64 transaction IDs (TIDs), and the application is responsible for tracking which TIDs are in use. The commands in the table move a transaction between three possible states: Free, Pending, or Committed (shown in Figure 3.1).

To create a new transaction with TID T, the application issues a `LogWrite` command with T as the first parameter. `LogWrite` records the data, size, and target location in a log, but does not modify the contents of the target location. After the first `LogWrite`, the state of the transaction changes from Free to Pending, indicating that the transaction is in progress but not committed. Additional calls

Table 3.3: **Multi-part atomic write commands** These commands allow the application to perform atomic and durable updates to the storage array. `LogWrite` returns a TID to the user that must be used on subsequent operations in the same atomic write.

Command	Description
`LogWrite(TID, file, offset, data, len, logfile, logoffset)`	Record a write to the log at the specified log offset. After commit, copy the data to the offset in the file.
`Commit(TID)`	Commit a transaction.

`Abort(TID)`
`Abort(TID, logfile, logoffset)`
	Cancel the transaction entirely, or perform a partial rollback to a specified point in the log.

`AtomicWrite(TID, file, offset, data, len, logfile, logoffset)`	Create and commit a transaction containing a single write.

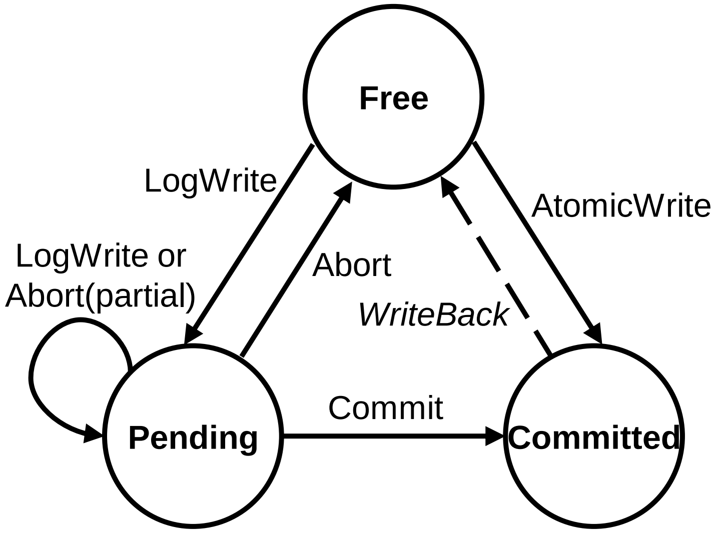


Figure 3.1: **Transaction state diagram** The system tracks the state of each transaction to guarantee that updates are atomic and durable.

to LogWrite add new writes to the transaction.

The writes in a transaction are not visible to other transactions until after commit. However, the transaction can see its own writes prior to commit by keeping track of the log offsets that it associates with each piece of data. After an initial log write for a particular piece of data, a transaction may update that data again before commit simply by writing to the correct log location.

To complete a transaction, the application issues Commit(T). The call to Commit assigns the transaction a commit sequence number inside the storage array that determines the commit order of this transaction relative to others. When the command completes, the transaction has logically committed, and the transaction moves to the COMMITTED state. If a failure should occur after a transaction logically commits but before the system writes the data back, then the recovery mechanism will replay the log to successfully complete the in-place updates.

The hardware can notify the application that the Commit is complete before the hardware copies the contents of log into their target locations, but during the commit process, reads and writes to the affected areas stall. This ensures that from the perspective of any application accessing the storage array, commit occurs atomically. When the copy is complete, the TID returns to FREE and the hardware notifies the application that the transaction finished successfully. At this point, it is safe to read the updated data from its target locations.

The application can also abort a transaction, freeing any log entries associated with it and returning it to FREE. Our model supports partial rollbacks of transactions by allowing the user to specify an Abort command with an offset into the log. The log offset acts as a savepoint: Any log entries starting from the log offset and going up through the most recent log entry log will be freed, effectively canceling those updates.

Our system provides flexibility by allowing the application to specify atomic write operations in multiple parts. However, this interface adds some overhead because each operation requires a separate IO request. To mitigate this cost, AtomicWrite combines LogWrite and Commit requests into a single request, allowing the system to quickly execute transactions that comprise a single write or to avoid the separate `Commit` when it can identify the final write in a transaction.

The system stores the logs as regular files in the storage array, and the logs may expand or shrink in size as the working sets of transactions demand. Conventional storage systems must allocate space for logs as well, but they often use separate disks to improve performance. Our system relies on the log being internal to the storage device, since our performance gains stem from utilizing the internal bandwidth of the storage array's independent memory banks.

The application manages log space by operating on the log files directly with POSIX file IO. The log file can be extended by writing past the end of the file with `write()`, and the log can be truncated with `ftruncate()`.

3.2.2 Design rationale

Our simple multi-part atomic write model strikes a balance between implementation complexity and functionality. Our model does not provide full ACID transactions, only atomicity and durability. In particular, our system does not provide isolation between transactions or any locking facilities to mediate access to shared data. The application must implement those if needed. However, our system does provide facilities to make implementing these features easier (e.g., updateable log entries) by letting the application access and manage the log space directly. Consequently, transactions may grow in size as needed and they see the results of their own previous but uncommitted updates. This is a key feature for supporting scalable ARIES-style transactions in MARS.

The algorithm our implementation uses to manage and commit transactions is simple. We use redo logging alone and always update the target location on commit (i.e., we use no-steal and force policies in the memory controllers). The high internal bandwidth of our storage array and the fast random access performance of NVMs minimizes the impact of using such a simple logging protocol. It also simplifies the hardware, since replaying the logs of committed transactions is sufficient for recovery. Finally, it avoids the remapping of addresses in hardware that a steal or no-force policy would require to hide uncommitted updates.

Offloading logging to the storage array has several performance benefits.

From the host's perspective, an atomic write operation requires no more external bandwidth than a normal write operation. Unlike traditional WAL schemes, we do not need to use a separate checkpoint process to update data in-place. Instead, we write data directly after commit as it is less expensive to wait for the IO to complete than it is to checkpoint and clean the logs in software. Similarly, it is faster to perform this write back than to avoid the extra write with a copy-on-write scheme because of the extra complexity required to manage the address space.

We could implement a more complex transaction model with conflict detection, locking, roll back, etc., but crafting a one-size-fits-all solution to those problems is not possible. Instead, we focus on using atomic writes to accelerate and simplify ARIES, which provides full-fledged ACID transactions for existing applications. In addition, atomic writes can be used as a building block for transactional updates to persistent data structures and key-value stores such as Memcached. Section 3.5 evaluates the benefits of our transactional model.

3.3 Related Work

Atomicity and durability are critical to storage system design, and system designers have explored many different approaches to providing these guarantees. These include approaches targeting disks, flash-based SSDs, and non-volatile main memories (i.e., NVMs attached directly to the processor) using software, specialized hardware, or a combination of the two. Below, we describe existing systems in this area and highlight the differences between them and the system we describe in this work.

3.3.1 Disk-based systems

Most disk-oriented systems provide atomicity and durability via software with minimal hardware support. Many systems use ARIES-style [MHL$^+$92] write-ahead logging to provide durability, atomicity, and to exploit the sequential performance that disks offer. Our system, unlike previous ones, implements write-ahead logging in hardware at the memory controllers. ARIES-style logging is ubiquitous in storage and database systems today.

Recent work on segment-based recovery [SB09] revisits the design of write-ahead logging for ARIES with the goal of providing efficient support for application-level objects. By removing LSNs on pages, segment-based recovery enables DMA or zero-copy IO for large objects and request reordering for small objects. Our system can take advantage of the same optimizations because the hardware manages logs without using LSNs and without modifying the format or layout of logged objects.

Traditional implementations of write-ahead logging are a performance bottleneck in databases running on parallel hardware. Aether [JPS$^+$10] implements a series of optimizations to lower the overheads arising from frequent log flushes, log-induced lock contention, extensive context switching, and contention for centralized, in-memory log buffers. These bottlenecks will be exacerbated by fast NVM-based storage, but our system eliminates them almost entirely. Because we offload logging to hardware, we remove lock contention and the in-memory log buffers. With fast storage and a customized driver, our system minimizes context switching and log flush delays.

Stasis [SB06] uses write-ahead logging to support building persistent data structures on disk. Stasis provides full ACID semantics and concurrency for constructing high-performance data structures such as hash tables and B-trees. It would be possible to port Stasis to use our atomic write support, but achieving good performance would require significant changes to its internal organization.

Our system provides atomicity and durability at the device level. The Logical Disk [dJKH93] provides a similar interface and presents a logical block interface based on atomic recovery units (ARUs) [GHKdJ96] – an abstraction for failure atomicity for multiple writes. Like our system, ARUs do not provide concurrency control. Unlike our system, ARUs do not provide durability, but they do provide isolation.

File systems including WAFL [HLM94] and ZFS [Cor] use shadow paging to perform atomic updates, and recent work on transactional support for flash-based SSDs [PRZ08, ONW$^+$11] relies on similar copy-on-write schemes. Although fast NVMs do not have the restrictions of disk or flash, the atomic write support in our system would help make these techniques more efficient. Recent work on BPFS [CNF$^+$09] extends shadow paging to work in systems that support finer-grain atomic writes. They target non-volatile main memory (see below), but our atomic write support could implement their scheme as well.

Researchers have provided hardware support atomicity in disks. Mime [CEJ$^+$92] is a high-performance storage architecture that uses shadow copies for this purpose. Mime offers sync and barrier operations to support ACID semantics in higher-level software. Like our system, Mime is implemented in the storage controller, but its implementation is more complex since it maintains a block map for copy-on-write, and maintains more metadata to keep track of the resulting versions.

3.3.2 Flash-based SSDs

Flash-based SSDs offer improved performance relative to disk, making latency overheads of software-based systems more noticeable. They also include complex controllers and firmware that uses remapping tables to provide wear-leveling and to manage flash's idiosyncrasies. The controller provides a natural opportunity to provide atomicity and durability guarantees, and several groups have done so.

Transactional Flash (TxFlash) [PRZ08] extends a flash-based SSD to implement atomic writes in the SSD controller. TxFlash leverages the fast random write performance and the copy-on-write nature of flash to perform atomic updates to multiple, whole pages with minimal overhead using "cyclic commit," a commit protocol that chains together log records using the out-of-band data associated with each flash page. Unlike SSDs based on flash, storage arrays of fast NVMs do not require a map, are not inherently copy-on-write, and are byte addressable. Consequently, our system logs and commits requests quite differently from TxFlash, and it allows arbitrarily sized and aligned requests.

Recent work from FusionIO [ONW$^+$11] proposes an atomic-write interface in a commercial flash-based SSD. Their system uses a log-based mapping layer in the drive's flash translation layer, but it requires that all the writes in one transaction be contiguous in the log. This prevents them from supporting multiple, simultaneous transactions.

3.3.3 Non-volatile main memory

The fast NVMs that our system targets are also candidates for non-volatile replacements for DRAM, potentially increasing storage performance dramatically. Using non-volatile main memory as storage will require atomicity guarantees as well, and several groups explored options in this space.

Recoverable Virtual Memory (RVM) [SMK$^+$93] provides persistence and atomicity for regions of virtual memory. It buffers transaction pages in memory and flushes them to disk on commit. RVM only requires redo logging because uncommitted changes are never written early to disk, but RVM also implements an in-memory undo log so that it can quickly revert the contents of buffered pages without rereading them from disk when a transaction aborts. Rio Vista [LC97] builds on RVM but uses battery-backed DRAM to make stores to memory persistent, eliminating the redo log entirely. Both RVM and Rio Vista are limited to transactions that can fit in main memory.

More recently, Mnemosyne [VTS11] and NV-heaps [CCA$^+$11] provide transactional support for building persistent data structures in byte-addressable, non-volatile memories. Both systems map NVMs attached to the memory bus into the application's address space, making it accessible by normal load and store instructions. Our atomic write hardware support could help implement a Mnemosyne- or NV-Heaps-like interface on a PCIe-attached storage device, but the details of the implementation would be very different.

3.4 Implementation

In this section, we present the details of the implementation of our interface described in Section 3.2, including how we issue commands to the array, how our software layer makes logging flexible and efficient, and how the hardware implements a distributed scheme for redo logging, commit, and recovery. We also discuss testing the system.

3.4.1 Software support

To make logging transparent and flexible, we leverage the existing software stack. First, we extend a user-space driver to implement our transaction API (see Section 3.2). In addition, we utilize the file system to manage the logs, exposing them to the user and providing an interface that lets the user dictate the layout of the log in storage.

User-space driver Our prototype SSD provides a highly-optimized (and unconventional) interface for accessing data [CME+12]. It provides a user-space driver that allows the application to communicate directly with the array via a private set of control registers, a private DMA buffer, and a private set of 64 tags that identify in-flight operations. To enforce file protection, the user space driver works with the kernel and the file system to download extent and permission data into Moneta, which then checks that each access is legal. As a result, accesses to file data do not involve the kernel at all in the common case. Modifications to file metadata still go through the kernel. Applications can use the new interface without modification, or even recompilation, since the library interposes on file access calls via LD_PRELOAD. The user space interface lets Moneta perform IO operations very quickly: 4 KB reads and writes execute in ~7 μs.

Our system uses this user space interface and includes the hardware permission checks on file accesses. We modify the user-space driver to implement our transaction API and provide each application with a private set of 64 virtual transaction IDs (TIDs), eliminating synchronization across applications. As a result, applications issue `LogWrite`, `Commit`, `Abort`, and `AtomicWrite` requests to the storage array from user space, avoiding costly interaction with the operating system.

File system managed logs Our system creates and manages the log through the file system and exposes the logs to the user. Our system uses two types of files to maintain a log for each transaction: a *log file* and a *metadata file*. The log file contains redo data for each update specified by the application. The metadata file records information about each update including the target location for the redo data upon transaction commit. Separating the metadata from the redo data allows applications to access the redo data in the same manner as accessing regular application data.

The user creates a log file and can extend or truncate the file, based on the application's log space requirements, using regular file IO. On `LogWrite` and `AtomicWrite` requests, the user specifies the location of a write into the log, which we call a *log offset*.

In order to have scalable transactions, the metadata must also be able to grow or shrink in size according to the working set demands. Our system stores log metadata in a metadata file that, unlike the log files, the application cannot modify. If the user could manipulate the metadata, the log space could become corrupted and unrecoverable. Even worse, the user might direct the hardware to update arbitrary storage locations, circumventing the protection of the OS and file system.

A trusted process, which we call the *metadata handler*, creates and manages metadata files. By communicating with this process, the user space driver can "install" and "remove" metadata files for the channel as needed. An install operation allocates a contiguous group of metadata entries in hardware on behalf of the application. As transactions execute, the hardware uses these metadata entries to store log metadata from `LogWrite` and `AtomicWrite` requests. When an application ends, the user space driver removes the metadata file, cleaning all metadata entries and releasing the metadata file back to the metadata handler.

To take advantage of the parallelism and internal bandwidth of Moneta, we require that data and log space be colocated at each memory controller. The user space driver ensures the data offset and log offset for `LogWrite` and `AtomicWrite` requests, when translated from logical to physical addresses, target the same mem-

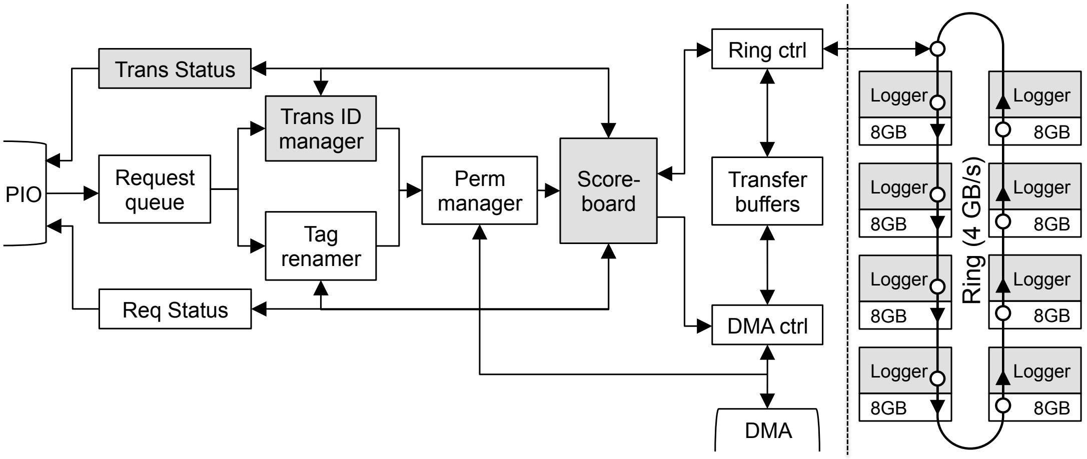


Figure 3.2: **SSD controller architecture** To support transactions, our system adds hardware support (gray boxes) to an existing prototype SSD. The main controller (to the left of the dotted line) manages transaction IDs and uses a scoreboard to track the status of in-flight transactions. Eight loggers perform distributed logging, commit, and recovery at each memory controller.

ory controller in the storage array.

3.4.2 Hardware support

The implementation of our atomic write interface divides functionality between two types of hardware components. The first is a logging module, which we call the *logger* (the gray boxes to the right of the dashed line in Figure 3.2), that resides at each of the system's eight memory controllers and handles logging for the local controller. The second is a set of modifications to the central controller (the gray boxes to the left of the dashed line in Figure 3.2) that orchestrates operations across the eight logging modules. Below, we describe the layout of the log and the components and protocols the system uses to coordinate logging, commit, and recovery.

Log structure Figure 3.3 shows an example log for a transaction at a logger. An entry in a transaction table points to an entry in a metadata file. Each metadata entry contains information about an entry in a log file and a pointer to the next

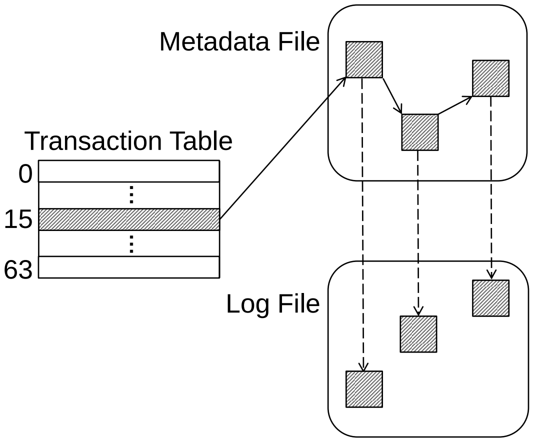


Figure 3.3: **Example log layout at a logger** The transaction table entry for transaction 15 contains the address for the first metadata entry in the log's linked list. Each metadata entry points to the next metadata entry in the linked list. It also points to the address of the block in the log file.

metadata entry.

The system stripes the log file across all 8 memory controllers in 8 KB chunks. The central controller routes log requests to the proper memory controllers, ensuring that the logger logs each 8 KB chunk of data at the same memory controller where the data it will update is located. This is essential for high performance because it enables each memory controller to write back in parallel, leveraging the large internal bandwidth at the memory controller.

When the metadata handler installs a metadata file, the hardware divides it into 32 B metadata entries. Each metadata entry contains information about a log entry. The logger allocates metadata entries to each `LogWrite` and `AtomicWrite` request, so the metadata entries for the same transaction may not be contiguous. Each metadata entry contains an address to the next metadata entry for the same transaction. Therefore, the log for a particular transaction is simply a linked list of metadata entries.

The system reserves a small portion (2 KB) of the storage at each memory controller for a transaction table. The transaction table stores the state for up to 64 transactions. Each entry in the transaction table includes the status of the

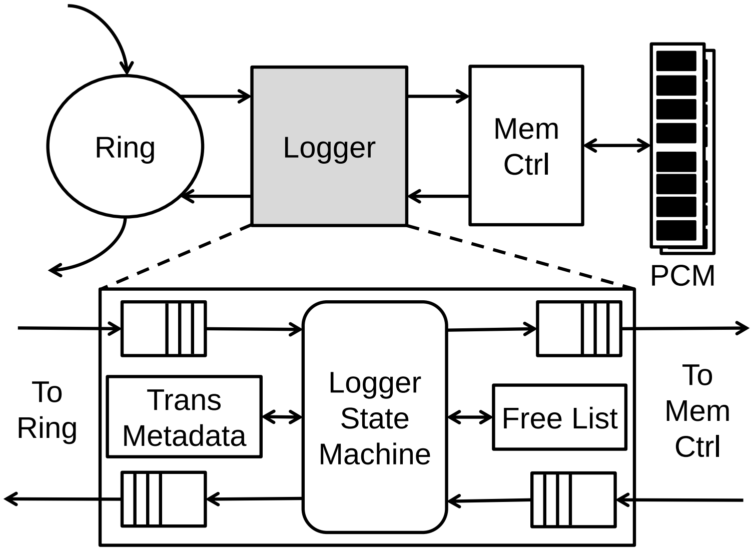


Figure 3.4: **Logger module** The logger sits between the ring and storage, allowing it to implement logging operations and issue its own requests to memory.

transaction, a sequence number, the address of the head metadata entry in the log, and the number of entries in the log.

Distributed logging Each of the eight memory controllers contains a logger module that independently performs logging, commit, and recovery operations and handles accesses to the 8 GB of NVM storage at the memory controller.

The logger implements `LogWrite`, `AtomicWrite`, `Commit`, `Abort`, and log recovery operations. Figure 3.4 shows the architecture of the logger and its relationship to the memory controller. The logger sits between the ring interface and the memory controller, allowing it to intercept operations and issue requests to the memory controller to manipulate log data and metadata.

Before an application can make a `LogWrite` or `AtomicWrite` request, it must first direct the metadata handler to install a metadata file. As mentioned above, the logger will divide the metadata file into metadata entries and the logger will use them to satisfy `LogWrite` and `AtomicWrite` requests. The logger maintains a free list of metadata entries for each channel in storage. When the logger divides the metadata file into metadata entries, it creates a linked list of metadata entries in storage by writing the pointer field for each metadata entry. The logger only maintains pointers to the head and tail of the free list for each channel and, therefore, can scale with the number of metadata entries.

To begin a new transaction, an application must have access to a previously created log file. As mentioned in Section 3.4.1, the application maintains the log file and specifies a log offset for each `LogWrite` or `AtomicWrite` request.

For each `LogWrite` request, the logger allocates a metadata entry, copies the data to the log offset, records the request information in the metadata entry, and then appends the metadata entry to the log.

For an `AtomicWrite` operation, the logger writes the data to the log and immediately marks the transaction COMMITTED, avoiding the extra delay required to coordinate across multiple memory controllers.

The logger implements `Commit` by waiting for all outstanding writes to the log area to complete and then marking the transaction as COMMITTED.

There are two kinds of `Abort` operations that the logger performs: *complete* and *partial*. For a complete `Abort` operation, the logger clears the transaction status and deallocates the metadata entries. On a partial `Abort`, the logger frees the metadata entries until the specified savepoint and makes the savepoint the new head of the transaction's metadata linked list.

A transaction is fully committed when all loggers have marked the transaction as COMMITTED in their transaction tables. The central controller (see next subsection) then directs each logger to apply their respective log. To apply the log, the logger reads each metadata entry in the log linked list. The transaction table indicates the metadata entry at the head of the linked list, as well as the number of entries in the list. For each metadata entry, the logger copies the redo data from the log offset to its destination address. During log application, the logger suspends other read and write operations to make log application appear atomic. At the end of log application, the logger deallocates the transaction's metadata entries. Since logging and data updates occur locally at each memory controller, logging and commit bandwidth scale with the number of controllers.

The central controller A single transaction may require the coordinate efforts of one or more memory controllers. The central controller (the left hand portion of Figure 3.2) coordinates the concurrent execution of `LogWrite`, `AtomicWrite`, `Commit`, `Abort`, and log recovery commands across the loggers. The central controller also handles `AtomicWrite` commands. If an `AtomicWrite` specifies a write that is within a stripe of 8 KB at a single memory controller, then the central controller sends the `AtomicWrite` directly to the target logger. Otherwise, the central controller breaks the `AtomicWrite` up into the appropriate `LogWrite` commands followed by a `Commit`. In the first case, the central controller avoids the extra latency to coordinate the commit across memory controllers. In either case, system avoids an extra IO request for an explicit `Commit`.

Three hardware components work together to implement transactional operations. First, the TID manager maps virtual TIDs from application requests to physical TIDs and tracks the transaction commit sequence number for the system. Second, the transaction scoreboard tracks the state of each transaction and enforces ordering constraints during commit and recovery. Finally, the transaction status table exports a set of memory-mapped IO registers that the host system interrogates during interrupt handling to identify completed transactions.

The central controller assigns a physical TID to incoming `LogWrite` and `AtomicWrite` requests, unless they have already received a physical TID from a previous request.

To perform a `LogWrite` the central controller breaks up requests along stripe boundaries, sends local `LogWrite`s to affected memory controllers, and awaits their completion. To maximize performance, our system allows multiple `LogWrite`s from the same transaction to be in-flight at once. If the `LogWrite`s are to disjoint areas, they will behave as expected. However, if they overlap, the results are unpredictable because parts of two requests may arrive at loggers in different orders. In those cases, the application can enforce an ordering by issuing a barrier command that will force outstanding `LogWrite` requests to finish before proceeding.

On `Commit`, the central controller increments the global transaction sequence number and broadcasts a commit command with the sequence number to the memory controllers that received `LogWrite`s. The memory controllers respond as soon as they have completed any outstanding `LogWrite` operations and have marked the transaction as committed. When the central controller receives all responses, it signals the loggers to begin applying the log and simultaneously notifies the application that the transaction has committed. Notifying the application before the loggers have finished applying the logs hides part of the log application latency. This is safe since only a memory failure (e.g., a failing NVM memory chip) can prevent log application from eventually completing. In that case, we assume that the entire storage device has failed and the data it contains is lost (see Section 3.4.3).

Implementation complexity Adding support for atomic writes to the baseline system required only a modest increase in complexity and hardware resources. The Verilog implementation of the logger required 1372 lines, excluding blank lines and comments. The changes to the central controller are hard to quantify. Once placed and routed on the FPGAs, adding the eight loggers and changing the central controller increased hardware consumption by 26%.

3.4.3 Recovery

Our system coordinates recovery operations in the kernel driver rather than in hardware to minimize complexity. There are two problems it needs to solve: The first is that some memory controllers may have marked a transaction as COMMITTED while others have not. In this case, the transaction must abort. Second, the system must apply the transactions in the correct order as given by their commit sequence numbers.

On boot, the driver scans the transaction tables at each memory controller to assemble a complete picture of transaction state across all the controllers. It identifies the TIDs and sequence numbers for the transactions that all loggers have marked as COMMITTED and sorts them by sequence number. The kernel then issues a kernel-only `WriteBack` command for each of these TIDs that triggers log replay at each logger. Finally, it issues `Abort` commands for all the other TIDs.

Once this is complete, the array is in a consistent state, and the driver makes the array available for normal use.

3.4.4 Testing and verification

To verify the atomicity and durability of our multi-part atomic write interface, we added hardware support to emulate system failure and performed failure and recovery testing. This presents a challenge since the DRAM that our prototype uses is volatile. To overcome this problem, we added support to force a reset of the system, which immediately suspends system activity. During system reset, we keep the memory controllers active to send refresh commands to the DRAM in order to emulate non-volatility. We assume the system includes capacitors to complete memory operations that the memory chips are in the midst of performing, just as many commercial SSDs do. To test recovery, we send a reset from the host while running a test, reboot the host system, and run our recovery protocol. Then, we run an application-specific consistency check to verify that no partial writes are visible.

We used two workloads during testing. The first workload consists of 16 threads each repeatedly performing an `AtomicWrite` to its own 8 KB region. Each write comprises a repeated sequence number that increments with each write. To check consistency, the application reads each of the 16 regions and verifies that they contain only a single sequence number and that that sequence number equals the last committed value. In the second workload, 16 threads continuously insert and delete nodes from our B+tree. After reset, reboot, and recovery, the application runs a function to verify the consistency of the B+tree.

We ran the workloads over a period of a few days, interrupting them periodically. The consistency checks for both workloads passed after every reset and recovery.

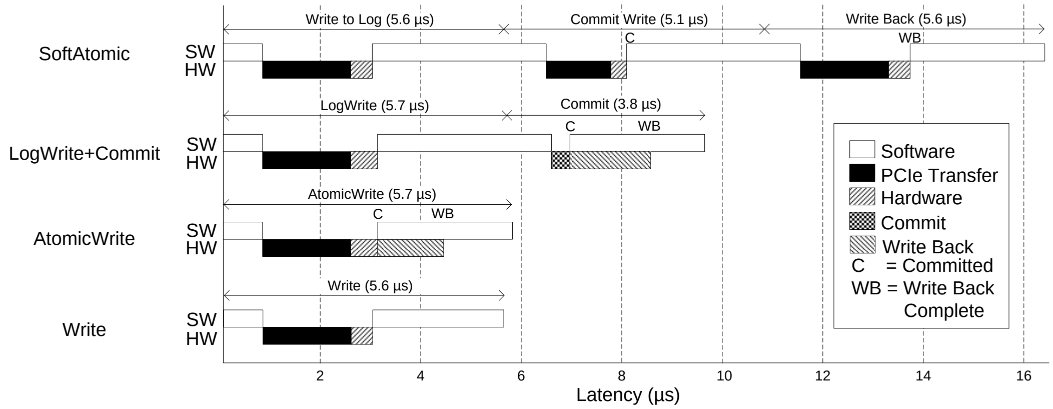


Figure 3.5: **Latency breakdown for 512 B atomic writes** Performing atomic writes without hardware support (top) requires three IO operations and all the attendant overheads. Using `LogWrite` and `Commit` reduces the overhead and `AtomicWrite` reduces it further by eliminating another IO operation. The latency cost of using `AtomicWrite` compared to normal writes is almost negligible.

3.5 Results

This section measures the performance of our multi-part atomic write primitive and evaluates its impact on MARS as well as other applications that require strong consistency guarantees. We first evaluate our system through microbenchmarks that measure the basic performance characteristics. Then, we present results for MARS relative to a traditional ARIES implementation, highlighting the performance improvement in a database setting. Finally, we show results for a set of three complex persistent data structures and MemcacheDB [Chu], a persistent key-value store for web applications.

3.5.1 Latency and bandwidth

Implementing atomic writes in hardware reduces the overhead of the multiphase write algorithms that applications traditionally use to write reliably to disk (e.g., writing a log entry, marking it with a commit record, and writing the data in-place).

Figure 3.5 shows the latencies of each stage of a 512 B atomic write implemented three different ways: Using multiple, synchronous non-atomic writes ("SoftAtomic"), using `LogWrite` followed by a `Commit` ("LogWrite+Commit"), and using `AtomicWrite`. As a reference, we include the latency breakdown for a normal write. For SoftAtomic we buffer writes in memory, flush the writes to a log, write a commit record, and then write the data in place. We used a modified version of XDD [xdd] to collect the data.

The figure shows the transitions between hardware and software and two different latencies for each operation. The first is the *commit latency* between command initiation and when the application learns that the transaction logically commits (marked with "C"). For applications using atomic writes to implement transactions (e.g., writing to a log), the commit latency is the critical latency. The second latency, the *write back latency* is from command initiation to the completion of the write back (marked with "WB"). At this point the TID becomes available for use again and the in-place updates finished.

The largest savings (41.4%) come from reducing the number of DMA transfers from three for SoftAtomic to one for the others (LogWrite+Commit takes two IO operations, but the `Commit` does not need a DMA). Using `AtomicWrite` to eliminate the separate `Commit` operation reduces latency by an additional 41.8%. Because an aligned 512 B access targets a single memory controller, the hardware can perform the atomic update at a single logger, eliminating the coordination overhead with the central controller.

Figure 3.6 plots the effective bandwidth (i.e., excluding writes to the log) for atomic writes ranging in size from 512 B to 512 KB. Our scheme increases throughput by between 2 and 3.8× relative to SoftAtomic. The data also show the benefits of `AtomicWrite` for small requests: transactions smaller than 4 KB achieve 92% of the bandwidth of normal writes in the baseline system.

Figure 3.7 shows the source of the performance improvement for multi-part atomic writes. It plots the total bytes read or written across all the memory controllers internally. For writes, internal and external bandwidth are the same. SoftAtomic achieves the same internal bandwidth because it saturates the PCIe bus, but roughly half of that bandwidth goes to writing the log. LogWrite+Commit

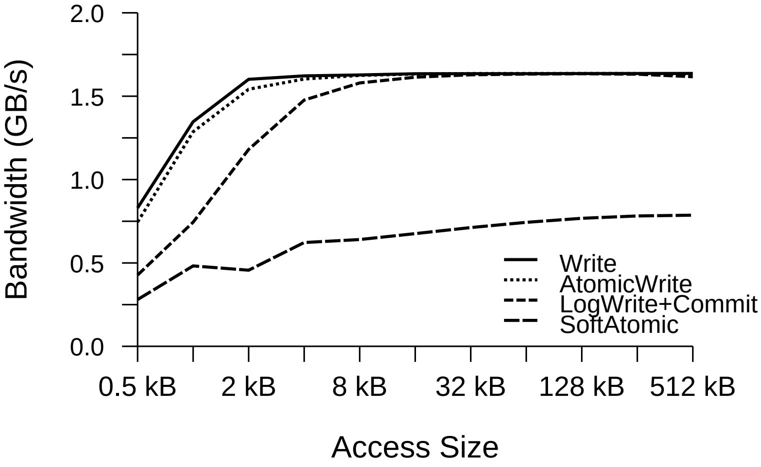


Figure 3.6: **Transaction throughput** By moving the log processing into the storage device, our system is able to achieve transaction throughput nearly equal to normal, non-atomic write throughput.

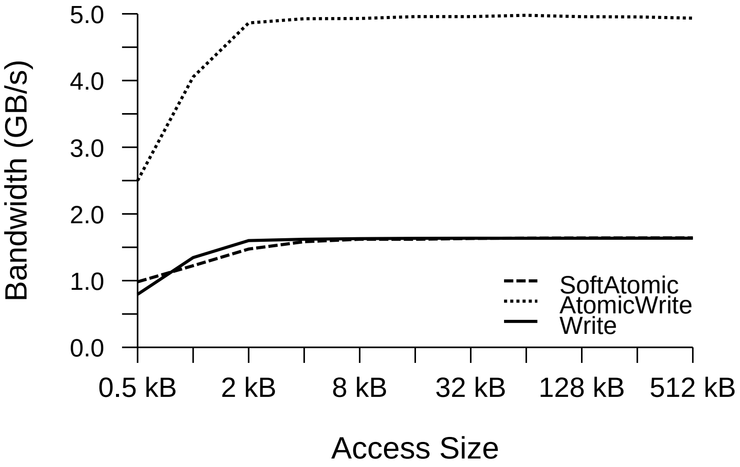


Figure 3.7: **Internal bandwidth** Hardware support for atomic writes allows our system to exploit the internal bandwidth of the storage array for logging and devote the PCIe link bandwidth to transferring useful data.

and `AtomicWrite` consume much more internal bandwidth (up to 5 GB/s), allowing them to saturate the PCIe link with useful data and better utilize the memory controllers.

3.5.2 MARS Evaluation

This section evaluates the benefits of MARS compared to a baseline implementation of ARIES. For this experiment, our benchmark transactionally swaps objects (pages) in a large database-style table.

The baseline implementation of ARIES performs the undo and redo logging required for steal and no-force. It includes a checkpoint thread that manages a pool of dirty pages, flushing pages to the storage array as the pool fills.

Our MARS implementation uses multi-part atomic writes to eliminate no-force and steal. The hardware implements a force policy at the memory controllers and we rely on the log to hold the most recent copy of an object prior to commit, giving us the benefits of a steal policy without requiring undo logging. Using a force policy in hardware eliminates the extra IO requests needed to commit and write back data. Removing undo logging and write backs reduces the amount of data sent to the storage array over the PCIe link by a factor of two.

Figure 3.8 shows the throughput, measured in transactions per second, for between 1 and 16 threads concurrently swapping objects of 4 KB, 16 KB, and 64 KB. The solid lines show the performance of MARS using atomic writes and the dashed lines show the performance of the baseline implementation of ARIES. For small transactions, where logging overheads are largest, our system outperforms ARIES by as much as 3.7×. For larger objects, the gains are smaller—3.1× for 16 KB objects and 3× for 64 KB. In these cases, ARIES makes better use of the available PCIe bandwidth, compensating for some of the overhead due to additional logs writes and write backs. MARS also scales better than ARIES: We see 1.6× speedup going from 4 threads to 8 threads for 4 KB objects, compared to a 0.5% performance loss for ARIES.

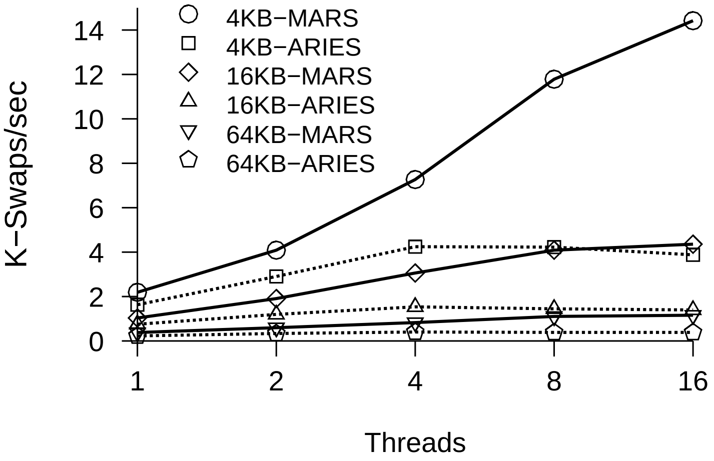


Figure 3.8: **Comparison of MARS and ARIES** Because fast NVMs have good random write performance, there is little benefit to no-force and steal transactions. With hardware support, MARS eliminates undo logging and write backs in order to maximize bandwidth and minimize resource contention.

3.5.3 Persistent data structure performance

We evaluate our system's impact on several light-weight persistent data structures designed to take advantage of our user space driver and transactional hardware support. The data structures include a hash table, a B+tree, and a large scale-free graph that supports "six degrees of separation" queries.

The hash table implements a transactional key-value store. It resolves collisions using separate chaining, and it uses per-bucket locks to handle updates from concurrent threads. Typically, a transaction requires only a single write to a key-value pair. But, in some cases an update requires modifying multiple key-value pairs in a bucket's chain. The footprint of the hash table is 32 GB, and we use 25 B keys and 1024 B values for this experiment. Each thread in the workload repeatedly picks a key at random within a specified range and either inserts or removes the key-value pair depending on whether or not the key is already present.

The B+tree also implements a 32 GB transactional key-value store. It caches the index, made up of 8 KB nodes, in memory for quick retrieval. To support a high degree of concurrency, it uses Bayer and Scholnick's algorithm [BS77] based on node safety and lock coupling. The B+tree is a good case study for our system because transactions can be complex: An insertion or deletion may cause splitting or merging of nodes throughout the height of the tree. Each thread in this workload repeatedly inserts or deletes a key-value pair at random.

Six Degrees operates on a large, scale-free graph representing a social network. It alternately performs Dijkstra's algorithm to find six-edge paths and modifies the graph by inserting or removing an edge. We use a 32 GB footprint for the undirected graph and store it in adjacency list format. Rather than storing a linked list of edges for each node, we use a linked list of edge pages, where each page contains up to 256 edges. This allows us to read many edges in a single request to the storage array. Each transactional update to the graph acquires locks on a pair of nodes and modifies each node's linked list of edges.

Figure 3.9 shows the performance for three implementations of each workload running with between 1 and 16 threads. The first implementation, "Unsafe," does not provide any durability or atomicity guarantees and represents an upper limit on performance. For all three workloads, adding ACID guarantees in software reduces performance by between 28 and 46% compared to Unsafe. For the B+tree and hash table, our atomic write support sacrifices just 13% of the performance of the unsafe versions on average. Six Degrees, on the other hand, sees a 21% performance drop with atomic writes because its transactions are longer and modify multiple nodes. Using atomic writes also improves scaling slightly. For instance, the `AtomicWrite` version of HashTable closely tracks the performance improvements of the Unsafe version, with only an 11% slowdown at 16 threads while the SoftAtomic version is 46% slower.

3.5.4 MemcacheDB performance

To understand the impact of hardware transactional support at the application level, we integrated our hash table into MemcacheDB [Chu], a persistent version of Memached [mem], the popular key-value store. The original Memcached uses a large hash table to store a read-only cache of objects in memory. MemcacheDB supports safe updates by using Berkeley DB to make the key-value

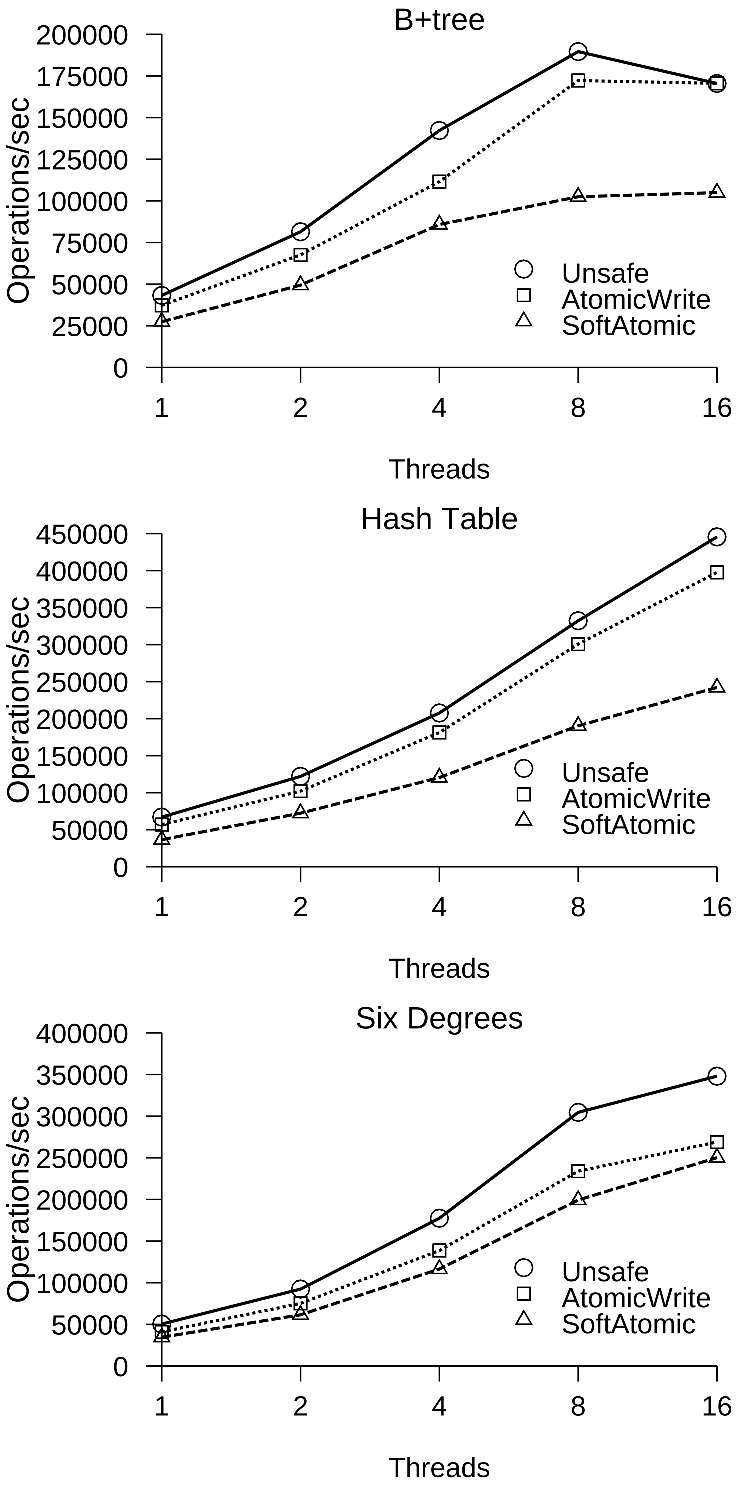


Figure 3.9: **Workload performance** Each set of lines compares the throughput of our B+tree (upper), hash table (middle), and Six Degrees (lower) workloads for Unsafe, `AtomicWrite`, and SoftAtomic versions as we scale the number of threads.

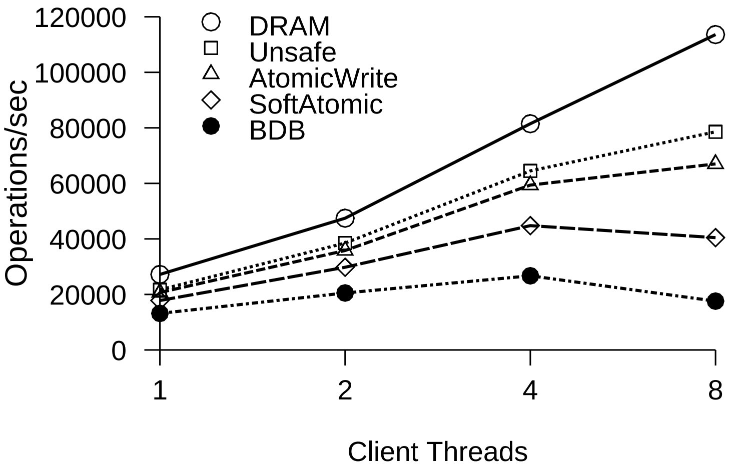


Figure 3.10: **MemcacheDB performance** Adding hardware support for atomicity increases performance by 1.7× for eight clients, and comes within 15% of matching the performance of an unsafe version that provides no durability.

store persistent. MemcacheDB uses a client-server architecture, and, for this experiment, we run it on a single computer acting as both clients and server.

Figure 3.10 compares the performance of MemcacheDB using our hash table as the key-value store (`AtomicWrite`) to versions that use volatile DRAM, a BDB database (labeled "BDB"), an in-storage key-value store without atomicity guarantees ("Unsafe"), and a SoftAtomic version. For eight threads, our system is 41% slower than DRAM and 15% slower than the Unsafe version. It is also 1.7× faster than the SoftAtomic implementation and 3.8× faster than BDB. Note that BDB provides many advanced features that add overhead but that MemcacheDB does not need and our implementation does not provide. Beyond eight threads, performance degrades because the application uses a single lock for updates.

3.6 Summary

Existing transaction mechanisms such as ARIES were designed to exploit the characteristics of disk, making them a poor fit for storage arrays of fast, non-volatile memories. This chapter presented a new WAL scheme, called MARS, designed for fast NVM-based storage. MARS provides the same set of features to the application as ARIES does but utilizes a novel multi-part atomic write operation that takes advantage of the parallelism and performance of an advanced SSD architecture. We demonstrated MARS and multi-part atomic writes in the Moneta prototype storage array. Compared to transactions implemented in software, our system increases effective bandwidth by up to 3.8× and decreases latency by 2.9×. When applied to MARS, multi-part atomic writes yield a 3.7× performance improvement relative to a baseline implementation of ARIES requiring both redo and undo logging of pages. Across a range of persistent data structures, multi-part atomic writes improve operation throughput by an average of 1.4×.

MARS demonstrates the benefits of redesigning existing transaction mechanisms for fast storage. It achieves high performance and can be integrated into established databases and other existing applications. This works well because MARS relies on a block device abstraction for storage, as do many applications built for disk. While this may be a good choice for PCIe-attached storage, it adds overhead and restricts the interface to storage that could otherwise be accessed just like main memory. In the next chapter, we consider a new abstraction for storage that appears on the processor's memory bus. This abstraction allows programmers to program persistent data in the same way they program volatile data. We explore the issues related to supporting such an abstraction so that access to data is fast, flexible, and safe.

Acknowledgments

This chapter contains material from "Providing Safe, User Space Access to Fast, Solid State Disks", by Adrian M. Caulfield, Todor I. Mollov, Louis Eisner, Arup De, Joel Coburn, and Steven Swanson, which appears in *ASPLOS '12: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems.* The dissertation author was the fifth investigator and author of this paper. The material in this chapter is copyright ©2012 by the Association for Computing Machinery, Inc. (ACM). Permission to

make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

This chapter contains material from "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories", by Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson, wwhich appears in *MICRO-43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. The dissertation author was the third investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

This chapter contains material from "From ARIES to MARS: Reengineering Transaction Management for Next-Generation, Non-Volatile Memories", by Joel Coburn, Trevor Bunker, Rajesh K. Gupta, and Steven Swanson, which will be submitted for publication. The dissertation author was the primary investigator and author of this paper.

Chapter 4

NV-heaps

In the previous chapter, we presented hardware support for transactions in a PCIe-attached storage array of fast, non-volatile memories and showed how existing transaction mechanisms can be redesigned to exploit such a storage architecture. With a DRAM-like interface and similar performance, these new memories can also be placed on the processor's memory bus, providing the system with direct access to fast storage. Direct access to storage enables programmers to build high-performance, persistent data structures in the same way they build volatile data structures. Instead of relying on untyped file-based IO operations, programmers can use the features of modern programming languages to interact with storage.

However, for these data structures to be useful, there must be safety guarantees against failures and programmer errors. This means that the system must prevent familiar bugs such as dangling pointers, multiple free()s, and locking errors. In addition, the system must prevent new types of hard-to-find pointer safety bugs that only arise with persistent objects. These bugs are especially dangerous since any corruption they cause will be permanent.

This chapter presents a lightweight, high-performance persistent object system called *Non-volatile Memory Heaps* (NV-heaps) that provides transactional semantics while preventing dangerous programming errors and providing a model for persistence that is easy to use and reason about. Section 4.1 makes the case for NV-heaps by arguing that existing systems are a poor fit for fast, non-volatile memories. In Section 4.2, we provide a system overview of NV-heaps and describe the guarantees it must provide for persistent data structures. Section 4.3 describes the implementation of NV-heaps, and Section 4.4 presents an evaluation of NV-heaps, showing both the overheads of the system and the performance relative to existing persistent object systems. Finally, Section 4.5 summarizes the benefits of NV-heaps in terms of both performance and safety in the presence of failures.

4.1 The Case for NV-heaps

The notion of memory-mapped persistent data structures has long been compelling: Instead of reading bytes serially from a file and building data structures in memory, the data structures would appear, ready to use in the program's address space, allowing quick access to even the largest, most complex persistent data structures. Fast, persistent structures would let programmers leverage decades of work in data structure design to implement fast, purpose-built persistent structures. They would also reduce our reliance on the traditional, un-typed file-based IO operations that do not integrate well with most programming languages.

Many systems (e.g., object-oriented databases) have provided persistent data structures and integrated them tightly into programming languages. These systems faced a common challenge that arose from the performance and interface differences between volatile main memory (i.e., DRAM) and persistent mass storage (i.e., disk): They required complex buffer management and de(serialization) mechanisms to move data to and from DRAM. Despite decades of work optimizing this process, slow disks ultimately limit performance, especially if strong consistency and durability guarantees are necessary. But, new non-volatile memory technologies, such as PCM and STTM, are poised to remove the disk-imposed limit on persistent object performance. With a DRAM-like byte-addressable interface and DRAM-like performance, these memories will come to reside on the processor's memory bus and will nearly eliminate the gap in performance between volatile and non-volatile storage.

Neither existing implementations of persistent objects nor the familiar tools we use to build volatile data structures are a good fit for these new memories. Existing persistent object systems are not suitable, because the gap between memory and storage performance drove many design decisions that shaped them. Recent work [CCM$^+$10, CDC$^+$10] has shown that software overheads from the operating system, file systems, and database management systems can squander the performance advantages of these memories. Removing these overheads requires significant reengineering of the way both the kernel and application manage access to storage.

Managing non-volatile memory like conventional memory is not a good solution either. To guarantee consistency and durability, non-volatile structures must meet a host of challenges, many of which do not exist for volatile memories. They must avoid dangling pointers, multiple free()s, memory leaks, and locking errors, but they also must avoid several new types of hard-to-find programming errors. For instance, pointers from non-volatile data structures into volatile memory are inherently unsafe, because they are meaningless after the program ends. The system must also perform some kind of logging if non-volatile structures are to be robust in the face of application or system failure. Trusting the average programmer to "get it right" in meeting these challenges is both unreasonable and dangerous for non-volatile data structures: An error in any of these areas will result in permanent corruption that neither restarting the application nor rebooting the system will resolve.

This thesis proposes a new implementation of persistent objects called *Non-volatile Memory Heaps* (NV-heaps). NV-heaps aim to provide flexible, robust abstractions for building persistent objects that expose as much of the underlying memory performance as possible. NV-heaps provide programmers with a familiar set of simple primitives (i.e., objects, pointers, memory allocation, and atomic sections) that make it easy to build fast, robust, and flexible persistent objects. NV-heaps avoid OS overheads on all common case access operations and protect programmers from common mistakes: NV-heaps provide automatic garbage collection, pointer safety, and protection from several novel kinds of bugs that non-volatile objects make possible. NV-heaps are completely self-contained, allowing the system to copy, move, or transmit them just like normal files.

In designing NV-heaps, our goals are to provide safe access to persistent objects, to make persistent objects easy to program, and to achieve high performance. To this end, our system has the following of properties:

1. **Pointer safety.** NV-heaps, to the extent possible, prevent programmers from corrupting the data structures they create by misusing pointers or making memory allocation errors.

2. **Flexible ACID transactions.** Multiple threads can modify NV-heaps concurrently. NV-heaps are robust against application and system failure.

3. **Familiar interface.** The programming interface for NV-heaps is similar to the familiar interface for implementing volatile data structures.

4. **High performance.** Access to the data in an NV-heap is as fast as possible relative to the speed of the underlying non-volatile memory.

5. **Scalability.** NV-heaps are designed to scale to very large (many gigabytes to terabytes) data structures.

In the next section, we provide a system overview of NV-heaps and describe its features in more detail.

4.2 NV-heaps: System Overview

The goal of NV-heaps is to make it easy to build and use robust, persistent data structures that can exploit the performance that emerging non-volatile, solid-state memories offer. To achieve this, NV-heaps provides an easy-to-use application-level interface to a persistent object system tailored to emerging non-volatile memories. The designs of previous persistent object systems [AH87, Cat94, BOS91, LLOW91, SKW92, WD94] focus on hiding disk latency rather than minimizing software overhead, making them a poor fit for these new memory technologies.

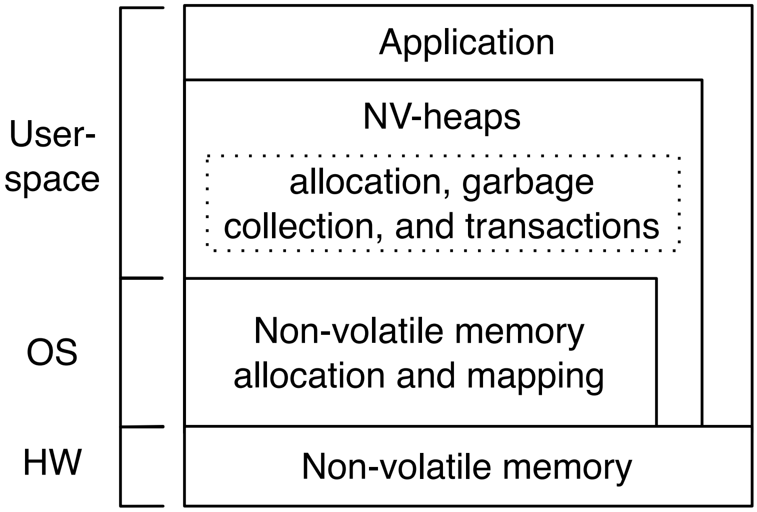

Application
NV-heaps
allocation, garbage collection, and transactions
Non-volatile memory allocation and mapping
Non-volatile memory
User-space
OS
HW

Figure 4.1: **The NV-heap system stack** This organization allows read and write operations to bypass the operating system entirely.

NV-heaps provide a small set of simple primitives: persistent objects, specialized pointer types, a memory allocator, and atomic sections to provide concurrency and guard against system or application failure. NV-heaps hide the details of locking, logging, and memory management, so building a data structure in an NV-heap is very similar to building one in a conventional, volatile heap.

To use an NV-heap, the application opens it by passing the NV-heap library a file name. The library maps the NV-heap directly into the application's address space without performing a copy, which is possible because the underlying memory is byte-addressable and appears in the processor's physical address space. Once the mapping is complete, the application can access the NV-heap's contents via a *root pointer* from which all data in the NV-heap is accessible.

Figure 4.1 shows the relationship between NV-heaps, the operating system, and the underlying storage. A key feature of NV-heaps is that they give the application direct access to non-volatile memory, eliminating operating system overhead in most cases and significantly increasing performance.

In this section, we make the case for the strong safety guarantees that NV-heaps provide. Then, we describe the transaction, pointer safety, performance, and usability features that NV-heaps include along with a code example using

NV-heaps. We discuss the differences between NV-heaps and existing persistent object systems throughout. Section 4.2 describes the implementation in detail.

4.2.1 Preventing programmer errors

Integrating persistent objects into conventional programs presents multiple challenges. Not only must the system (or the programmer) maintain locking and memory allocation invariants, it must also enforce a new set of invariants on which objects belong to which region of memory. Potential problems arise if one NV-heap contains a pointer into another NV-heap or into the volatile heap. In either case, when a program re-opens the NV-heap, the pointer will be meaningless and potentially dangerous. Furthermore, violating any of these invariants results in errors that are, by definition, persistent. They are akin to inconsistencies in a corrupt filesystem.

If persistent objects are to be useful to the average programmer, they must be fast *and* make strong safety guarantees. Providing a low-level interface and expecting the programmer to "get it right" has proven to be a recipe for bugs and unreliability in at least two well-known domains: Memory management and locking disciplines. In both of those instances, there is a program-wide invariant (i.e., which code is responsible for `free()`ing an object and which locks protect which data) that the source code does not explicitly describe and that the system does not enforce. A persistent object system must contend with both of these in addition to the constraints on pointer usage in NV-heaps.

To understand how easy it is to create dangerous pointers in NV-heaps, consider a function

```
Insert(Object * a, List<Object> * l) ...
```

that inserts a pointer to a non-volatile object, `a`, into a non-volatile linked list, `l`. The programmer must ensure that `a` and `l` are part of the same non-volatile structure, but there is no mechanism to enforce that constraint since it is not clear whether `a` is volatile or non-volatile or which NV-heap it belongs to. One incorrect call to this function can corrupt `l`: It might, for instance, end up containing a pointer from an NV-heap into volatile memory. In either case, if we move `l` to another system or restart the program, `l` is no longer safe to use: The pointer to object `a` that the list contains has become a “wild” pointer.

There is also real-world evidence that non-volatile data structures are difficult to implement correctly. Microsoft Outlook stores mail and other personal information in a pointer-based Personal Folder File (PFF) file format [Met]. The file format is complex enough that implementing it correctly proved difficult. In fact, Microsoft eventually released the “Inbox Repair Tool” [mic] which is similar to `fsck`-style tools that check and repair file systems.

Another system, BPFS [CNF+09], highlights what it takes to build a robust non-volatile data structure on top of raw non-volatile memory. BPFS implements a transactional file system directly atop the same kind of byte-addressable non-volatile memories that NV-heaps target. BPFS uses carefully designed data structures that exploit the file system’s tree structure and limited set of required operations to make transactional semantics easy to implement in most cases. In doing so, however, it enforces stringent invariants on those structures (e.g., each block of data has only a single incoming pointer) and requires careful reasoning about thread safety, atomicity, and memory access ordering. BPFS is an excellent example of what skilled programmers can accomplish with non-volatile memories, but average users will, we expect, be unwilling (or unable) to devise, enforce, and reason about such constraints. NV-heaps remove that burden, and despite additional overheads, they still provide excellent performance (see Section 4.4).

Existing systems that are similar to NV-heaps, such as Rio Vista [LC97] and Recoverable Virtual Memory (RVM) [SMK+93], provide direct access to byte addressable, non-volatile memories and let the programmer define arbitrary data structures. But these systems do not offer protection from memory allocation errors, locking errors, or dangerous non-volatile pointers. Concurrent work on a system called Mnemosyne [VTS11] goes further and provides a set of primitives for operating on data in persistent regions. It would be possible to implement the key features of NV-heaps using these primitives. Systems that target disk-based storage have either implemented persistent objects on top of a conventional database (e.g., the Java Persistence API [BO06]) or in a specialized object-oriented database [AH87, BOS91, Cat94, LLOW91, SKW92, WD94]. In these systems, the underlying storage system enforces the invariants automatically, but they extract a heavy cost in terms of software overhead.

4.2.2 Transactions

To make memory-mapped structures robust in the face of application and system failures, the system must provide useful, well-defined guarantees about how and when changes to a data structure become permanent. NV-heaps use ACID transactions for this purpose because they provide an elegant method for mediating access to shared data as well as robustness in the face of failures. Programmer-managed locks cannot provide that robustness. Recent work on (volatile) transactional memory [HM93, ST95, HWC+04, BDLM07, SATH+06, HLMS03, HF03] demonstrates that transactions may also be easier to use than locks in some cases.

Systems that provide persistence, including persistent object stores [LLOW91, LAC+96, SKW92, WD94], some persistence systems for Java [BO06, MZB00], Rio Vista [LC97], RVM [SMK+93], Mnemosyne [VTS11], Stasis [SB06], Argus [Lis88], and QuickSilver [HMSC87], provide some kind of transactions, as do relational databases. However, the type of transactions vary considerably. For example, Stasis provides page-based transactions using write-ahead logging, a technique inspired by databases [MHL+92]. Mnemosyne also uses write-ahead logging, but operates at the word granularity. RVM and Rio Vista provide transactions without isolation, and RVM provides persistence guarantees only at log flushes. Single-level stores have taken other approaches that include checkpoints [SA02] and explicitly flushing objects to persistent storage [Sol96].

4.2.3 Referential integrity

Referential integrity implies that all references (i.e., pointers) in a program point to valid data. Java and other managed languages have demonstrated the benefits of maintaining referential integrity. They are able to avoid memory leaks, "wild" pointers, and the associated bugs.

In NV-heaps, referential integrity is more important and complex than in a conventional system. Integrity problems can arise in three ways, and each requires a different solution.

Memory allocation NV-heaps are subject to the memory leaks and pointer bugs that all programming systems face. Memory leaks, in particular, are more pernicious in a non-volatile setting. Once a region of storage leaks away, reclaiming it is very difficult. Preventing such problems requires some form of automatic garbage collection.

NV-heaps use reference counting, which means that space is reclaimed as soon as it becomes dead and that there is never a need to scan the entire NV-heap. The system avoids memory leaks due to cycles by using a well-known technique: weak pointers that do not affect reference counts. Several other garbage collection schemes [KW93, ONG93] for non-volatile storage have been proposed, and integrating a similar system into NV-heaps is a focus of our ongoing work.

Previous systems have taken different approaches to memory management. Non-volatile extensions to Java [ADJ$^+$96, MZB00] provide garbage collection, but Rio Vista [LC97], RVM [SMK$^+$93], and Mnemosyne [VTS11] do not. Java's persistence API [BO06] requires the programmer to specify a memory management policy via flexible (and potentially error-prone) annotations. Object-oriented databases have taken a range of approaches from providing simple garbage collection [BOS91] to allowing applications to specify complex structural invariants on data structures [AH87].

Volatile and non-volatile pointers NV-heaps provide several new avenues for creating unsafe pointers because they partition the address space into a volatile memory area (i.e., the stack and volatile heap) and one or more NV-heaps.

This partitioning gives rise to four new types of pointers: Pointers within a single NV-heap (*intra-heap* NV-to-NV pointers), pointers between two NV-heaps (*inter-heap* NV-to-NV pointers), pointers from volatile memory to an NV-heap (V-to-NV pointers), and pointers from an NV-heap to volatile memory (NV-to-V pointers).

Ensuring referential integrity in NV-heaps requires that the system obey two invariants. The first is that there are no NV-to-V pointers, since they become meaningless once the program ends and would be unsafe the next time the program uses the NV-heap.

The second invariant is that there are no inter-heap NV-to-NV pointers. Inter-heap pointers become unsafe if the NV-heap that contains the object is not available. Inter-heap pointers also complicate garbage collection, since it is impossible to tell if a given location in an NV-heap is actually dead if a pointer in another (potentially unavailable) NV-heap may refer to it.

NV-heaps enforce these invariants via a simple dynamic type system. Each pointer and each object carries an identifier of the heap (NV-heap or volatile heap) that it belongs to. Mismatched assignments are a run-time error. As far as we know, NV-heaps are the first system to explicitly identify and prohibit these types of dangerous pointers. Rio Vista [LC97] and RVM [SMK$^+$93] make no attempts to eliminate any of these pointer types. Also, full-fledged persistent object systems such as ObjectStore do not guard against these dangerous pointers, leaving the system and underlying database vulnerable to corruption [HAG99]. However, other systems effectively eliminate dangerous pointers. JavaCard [All09], a subset of Java for the very constrained environment of code running on a smart card, makes almost all objects persistent and collects them in a single heap. In Java's persistence API [BO06] the underlying database determines which NV-to-NV pointers exist and whether they are well-behaved. It prohibits NV-to-V pointers through constraints on objects that can be mapped to rows in the database.

Closing NV-heaps Unmapping an NV-heap can also create unsafe pointers. On closing, any V-to-NV pointers into the NV-heap become invalid (but non-null). Our implementation avoids this possibility by unmapping NV-heaps only at program exit, but other alternatives are possible. For instance, a V-to-NV pointer could use a proxy object to check whether the NV-heap it points into is still open.

4.2.4 Performance and scalability

NV-heaps provide common case performance that is close to that of the underlying memory for data structures that scale up to the terabyte range. All common case operations (e.g., reading or writing data, starting and completing transactions) occur in user space. NV-heaps make system calls to map themselves into the application's address space and to expand the heap as needed. Entering the OS more frequently would severely impact performance since system call overheads are large (e.g., 6 μs for a 4 KB read on our system) compared to memory access time.

The result is a system that is very lightweight compared to previous persistent object systems that include sophisticated buffer management systems to hide disk latency [CDG$^+$90, JLR$^+$94, LLOW91, SKW92, WD94] and/or costly serialization/deserialization mechanisms [BO06]. Unlike these systems, NV-heaps operate directly on non-volatile data that is accessible through the processor-memory bus, thereby avoiding the operating system. Rio Vista [LC97] is similar to NV-heaps, since it runs directly on battery-backed DRAM, but it does not provide any of the safety guarantees of NV-heaps. Mnemosyne [VTS11] also provides direct access to fast, non-volatile memories, but providing the level of safety in NV-heaps requires more effort.

We have designed NV-heaps to support multi-terabyte data structures, so we expect that the amount of non-volatile storage available in the system may be much larger than the amount of volatile storage. To allow access to large structures, NV-heaps require a fixed, small amount of volatile storage to access an NV-heap of any size. NV-heaps also ensure that the running time of operations, including recovery, are a function only of the amount of data they access, not the size of the NV-heap.

4.2.5 Ease of use

To be useful, NV-heaps need to be easy to use and interact cleanly with each other and with existing system components. NV-heaps also need to make it clear which portions of a program's data are non-volatile and which NV-heap they belong to.

NV-heaps exist as ordinary files in a file system in a manner similar to previous persistent object store implementations [SKW92, WD94]. Using files for NV-heaps is crucial because the file system provides naming, storage management, and access control. These features are necessary for storage, but they are not needed for systems like [QKF$^+$09a, QSR09, ZZYZ09] that simply use non-volatile memories as a replacement for DRAM.

Like any file, it is possible to copy, rename, and transmit an NV-heap. This portability means that NV-heaps must be completely self-contained. The prohibition against NV-to-V and inter-heap NV-to-NV pointers guarantees this isolation. An additional feature, relative pointers (described in Section 4.3), provides (nearly) zero-cost pointer "swizzling" and makes NV-heaps completely relocatable within an application's address space.

We chose to make NV-heaps self-contained to make them easier to manage with existing file-based tools. However, this choice means that NV-heaps do not natively support transactions that span multiple NV-heaps. Implementing such transactions would require that NV-heaps share some non-volatile state (e.g., the "committed" bit for the transaction). The shared state would have to be available to both NV-heaps at recovery, which is something that self-contained NV-heaps cannot guarantee. It is possible to move objects between non-volatile data structures, but those structures need to reside in the same NV-heap.

While NV-heaps make it easy to implement non-volatile data structures, they do not provide the "orthogonal persistence" that persistent object stores [LLOW91, WD94, SKW92] and some dialects of Java [DKM10, MZB00, ADJ$^+$96] provide. Orthogonal persistence allows programmers to designate an existing pointer as the "root" and have all objects reachable from that pointer become *implicitly* persistent regardless of their type. This is an elegant abstraction, but using reachability to confer persistence leads to several potential problems. For instance, the programmer may inadvertently make more data persistent than intended. In addition, the abstraction breaks down for objects that cannot or should not be made persistent, such as an object representing a network connection, a file descriptor, or a secret key. Finally, it is possible for a single object to be reachable from two roots, leading to the confusing situation of multiple copies of the same object in two different persistent structures.

NV-heaps provide an alternative model for persistence. Each NV-heap has a designated root pointer, and everything reachable from the root is persistent and part of the same NV-heap. The difference is that the program *explicitly* creates a persistent object in the NV-heap and attaches it to another object in the heap with a pointer. This does not prevent all errors (e.g., it is still possible to inappropriately store a file descriptor in an NV-heap), but it requires that the programmer explicitly add the data to the NV-heap. It also prevents a single object from being part of two NV-heaps. This model for persistence is similar to what is imposed by the Thor [LAC$^+$96] persistent object store, but Thor does so through a type-safe database programming language.

4.2.6 Example

The code in Figure 4.2 provides an example of how a programmer can create a non-volatile data structure using NV-heaps. The code removes the value `k` from a linked list. Declaring the linked list class as a subclass of `NV_Object` marks it as non-volatile. The `DECLARE_POINTER_TYPES`, `DECLARE_MEMBER`, and `DECLARE_PTR_MEMBER` macros declare the smart pointer types for NV-to-NV and V-to-NV pointers (`NVList::NV_Ptr` and `NVList::V_Ptr`, respectively) and declare two fields. The declarations generate private fields in the class and public accessor functions (e.g., `get_next()` and `set_next()`) that provide access to data and perform the appropriate logging and locking operations.

The program uses `NVHOpen()` to open an NV-heap and then calls `GetRoot()` to retrieve the root object of the NV-heap, in this case a list of integers. The program stores the pointer to the linked list as a NVList::V_Ptr, which is a volatile

```
class NVList : public NVObject {
  DECLARE_POINTER_TYPES(NVList);
public:
  DECLARE_MEMBER(int, value);
  DECLARE_PTR_MEMBER(NVList::NVPtr, next);
};

void remove(int k)
{
  NVHeap * nv = NVHOpen("foo.nvheap");
  NVList::VPtr a =
            nv->GetRoot<NVList::NVPtr>();
  AtomicBegin {
    while(a->get_next() != NULL) {
      if (a->get_next()->get_value() == k) {
        a->set_next(a->get_next()->get_next());
      }
      a = a->get_next();
    }
  } AtomicEnd;
}
```

Figure 4.2: **NV-heap example** A simple NV-heap function that atomically removes all links with value `k` from a non-volatile linked list.

pointer that will only exist for the duration of the function call. The `AtomicBegin` operation starts a transaction. When the atomic section is complete (as indicated by the `AtomicEnd` label), the NV-heap attempts to commit the changes. If the transaction fails or if the system crashes, it will roll the operations back to restore the list to its original state.

In the next section we describe the implementation of the NV-heaps library in detail.

4.3 Implementing NV-heaps

Two considerations drove our implementation of NV-heaps: The need for strong safety guarantees and our goal of maximizing performance on fast, non-volatile memories. We implemented NV-heaps as a C++ library under Linux. The system is fully functional running on top of a RAM disk backed by DRAM. Below, we describe the technologies that NV-heaps target and the support they require from the OS and hardware. Then we describe our implementations of memory management, reference safety, and transactions. Finally, we discuss the storage overheads and how we validated our implementation.

4.3.1 Fast, byte-addressable non-volatile memories

NV-heaps target solid-state memory technologies that present a DRAM-like interface (e.g., via LPDDR [jed09]) and achieve performance within a small factor of DRAM. To evaluate NV-heaps we consider two advanced non-volatile memories: phase change memory (PCM) and spin-torque transfer memory (STTM).

PCM stores data as the crystalline state of a chalcogenide layer [Bre08] and has the potential to become a viable main memory technology as DRAM's scaling falters [LIMB09, QSR09, ZZYZ09]. PCM may also eventually surpass flash memory in density according to the ITRS [ITR09]. The analysis in [LIMB09] provides a good characterization of PCM's performance and power consumption.

STTM stores bits as a magnetic orientation of one layer of a magnetic tunnel junction [DSPE08]. We assume 22nm STTM technology and base our estimates for performance on published papers [TKM$^+$07, KTM$^+$08] and discussions with industry.

PCM and, to a lesser extent, STTM, along with most other non-volatile memories require some form of wear management to ensure reasonable device lifetime. Many wear-leveling schemes are available [DAR09, LIMB09, ZZYZ09, CL09, QKF$^+$09a] and some can provide excellent wear-leveling at the memory controller level for less than 1% overhead. NV-heaps (like BPFS [CNF$^+$09]) assume that the system provides this service to all the applications that use the storage.

4.3.2 System-level support

NV-heaps rely on a few simple facilities provided by the system. To the file system, NV-heaps appear as normal files. To access them efficiently, the file system should be running on top of the byte-addressable, non-volatile memory that appears in the CPU's physical address space. To open an NV-heap, the system uses `mmap()`. Normally, the kernel copies `mmap()`'d data between a block device and DRAM. In a system in which byte-addressable, non-volatile memories appear in the processor's address space, copying is not necessary. Instead, `mmap()` maps the underlying physical memory pages directly into the application's virtual address space. In our kernel (2.6.28), the brd ramdisk driver combined with ext2 provides this capability.

The second requirement is a mechanism to ensure that previous updates to non-volatile storage have reached the storage and are permanent. For memory-mapped files, `msync()` provides this functionality, but the system call overhead is too high for NV-heaps. Instead, NV-heaps rely on architectural support in the form of the atomic 8-byte writes and epoch barriers developed for BPFS [CNF$^+$09] to provide atomicity and consistency. Epoch barriers require small changes to the memory hierarchy and a new instruction to specify and enforce an ordering between groups of memory operations. BPFS also provides durability support by incorporating capacitors onto the memory cards to allow in-progress operations to finish in the event of a power failure. We assume similar hardware support.

4.3.3 Memory management

The NV-heap memory management system implements allocation, automatic garbage collection, reference counting, and pointer assignments as simple, fixed-size ACID transactions. These basic operations form the foundation on which we build full-blown transactions. The memory management system also provides the support required to reload NV-heaps.

Atomicity and Durability The allocator uses fixed-size, non-volatile, redo-logs called *operation descriptors* (similar to the statically-sized transactions in [ST95]) to provide atomicity and durability for memory allocation and reference-counted pointer manipulation. There is one set of operation descriptors per thread and the design ensures that a thread only ever requires one of each type of descriptor for all operations. Epoch barriers ensure that the descriptors are in a consistent state before the operation logically commits.

```
void RCAssign(RefCountPtr& dstPtr, RefCountPtr& object) {
  object->Lock();
  /* Set up the operation descriptor */
  opDescriptor->dstPtr = &dstPtr;
  opDescriptor->newCount = object->refCount + 1;
  EpochBarrier();
  opDescriptor->valid = true;
  EpochBarrier();
  /* Apply the operation descriptor */
  *(opDescriptor->dstPtr) = opDescriptor->object;
  opDescriptor->object->refCount = opDescriptor->newCount;
  EpochBarrier();
  opDescriptor->valid = false;
  EpochBarrier();
  object->Unlock();
}
```

Figure 4.3: **Pseudo-code for assignment to a reference counted pointer** This code uses a two-phase commit protocol to atomically and durably assign a new value to a reference counting pointer and update the reference count on the object.

Figure 4.3 contains the pseudo-code for atomically assigning the address of a reference counted object to a reference counting pointer that is initially NULL. The allocator uses a simple two-phase protocol to provide atomicity and durability. It uses a non-volatile operation descriptor to record the changes required to perform the assignment. In this case, the descriptor contains the address of the pointer receiving the new value, the address of the reference counted object, and the new value of the object's reference count.

To gather this data, the code starts by recording the address of the object. It then acquires a lock stored in the object that will prevent other threads from concurrently modifying its reference count and/or destroying it. Once it has gathered the rest of the data, the code uses an epoch barrier to guarantee that the data is recorded in non-volatile memory. Then, it sets a valid bit and issues another epoch barrier. At this point, the assignment has logically completed. It then "plays" the operation descriptor by performing the necessary assignments. Once the assignments are complete, it issues a third epoch barrier and then marks the descriptor invalid. In the case of one or more system failures, it can replay the assignments as many times as needed.

Concurrency To provide support for concurrent accesses to objects, each persistent object contains a lock that protects its reference count. The transaction system protects the other data in the object.

Locks are volatile by nature because they only have meaning during runtime. Therefore, to operate correctly, all locks must be released after a system failure. This could necessitate a scan of the entire storage array, violating our scalability requirement.

We avoid this problem by using *generational locks*: We associate a current *generation number* with each NV-heap, and the system increments the generation number when the NV-heap is reloaded. A generational lock is an integer. If the integer is equal to the current generation number, a thread holds it, otherwise, it is available. Therefore, incrementing the NV-heap's generation instantly releases all of its locks.

Allocation The allocator uses per-thread free lists and a shared global free list to reduce contention. On allocation, if a thread's free list does not contain a suitable memory block, the thread acquires a lock on the global free list and uses its operation descriptor to complete the allocation. If free space for the allocation is not available in the global free list, the NV-heap library expands the file that holds the NV-heap and maps that storage into the address space (similar to `sbrk()` in a conventional volatile memory allocator).

During allocation, storage must atomically move from control of the allocator to control of the code that requested the allocation to prevent orphaned data. The default behavior of a statement like

```
Foo * a = new Foo();
```

in C++ does not allow this, since there is a moment between the allocation and the assignment when the allocated storage would be lost if the system crashed. To avoid this, NV-heaps provide a static allocator method, `NVNew()`, which allocates storage, calls an object's constructor if one exists, and performs the assignment to the requester's pointer all as part of one atomic operation.

A useful side-effect of `NVNew()` is that the allocate-assign operation can atomically allocate and append a link in a singly-linked list. Our transaction implementation uses this ability to, for instance, atomically append entries to a non-volatile write log.

Prior work in memory management for multi-core systems played a part in the design of the NV-heap allocator. Memory allocators such as Hoard [BMBW00] use a global heap and per-thread heaps to efficiently support parallel applications. McRT-Malloc is a memory allocator designed specifically for a software transactional memory system in a multi-core environment [HSATH06].

Deallocation When the reference counting system discovers that an object is dead, it deallocates the storage. The deallocation routine atomically calls the destructor, deallocates the memory, and sets the requester's pointer to NULL. Deallocation is a potentially complex process since the destructor may cause the reference counts on other objects to go to zero, necessitating their destruction as well. NV-heaps must be able to restart this recursive destruction process in the case of a system failure.

To implement atomic recursive destruction, the NV-heap records the top level object to be destroyed in the *root deletion* operation descriptor and then calls its destructor. When the destructor encounters a pointer to another object, it checks that object's reference count. If the reference count is one (i.e., this is the last pointer to the object), it performs that deallocation using the *non-root deletion* descriptor. This means that it calls the object's destructor, and when the destructor completes, it atomically deallocates the memory and sets the pointer to NULL. If the reference count is greater than one, it atomically decrements the reference count and sets the pointer to NULL.

This algorithm provides the useful guarantee that, if a destructor finds a non-null pointer, it must point to a live object, albeit one that may have been partially destroyed. However, even if the object has been partially destroyed,

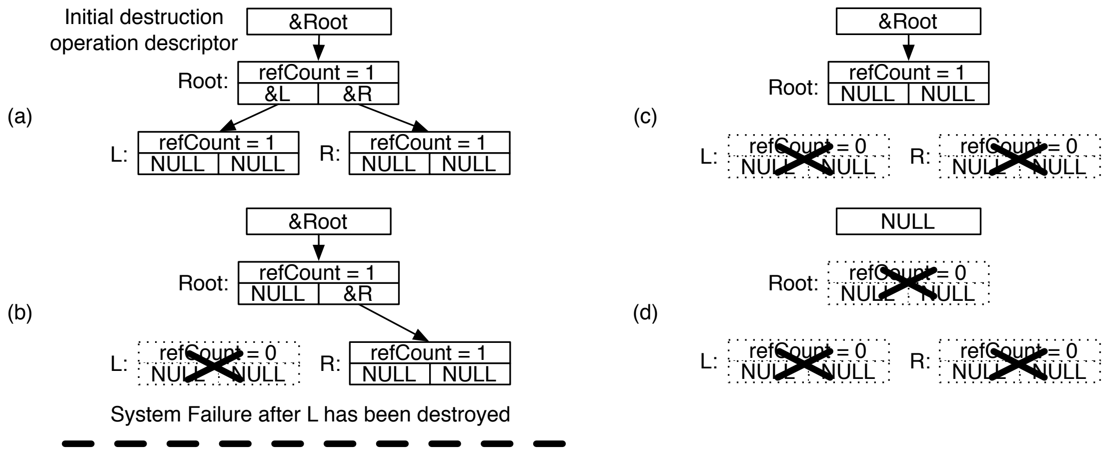


Figure 4.4: **Restartable object destruction** Durable, atomic reference counting requires being able to restart the recursive destruction of a potentially large numbers of objects. In this example the root node of a tree is destroyed and triggers the deletion of the entire tree. After a system failure, the process resumes and completes during the recovery process.

calling the destructor is still safe: The non-null pointers in the object point to live objects and the NULL pointers have already been properly disposed of.

If a deletion causes a further recursive deletion, the non-root descriptor is reassigned to that deletion. The combination of the root and non-root descriptors provide a snapshot of the recursive deletion process that 1) requires only two descriptors regardless of recursion depth and 2) allows the process to restart in the case of failure.

On recovery, the NV-heap processes the non-root descriptor first to restore the invariant that all non-null pointers in the structure point to valid objects. It then restarts the root deletion, which will try to perform the same set of recursive destructions. It will encounter NULL pointers up until the point of the system failure and then resume deletion where it was interrupted.

Figure 4.4 shows the algorithm in action. The system has just begun to delete a tree rooted at `Root` since the last pointer to it has just been set to NULL by the application. In (a), the operation descriptor for the starting point for the deletion holds the address of `Root`. The destruction proceeds by destroying the left child, `L`, and setting the left child pointer in `Root` to NULL.

At this point (b), the system fails. During recovery, it finds a valid operation descriptor and restarts the delete. Since the original destruction of `L` was atomic, the pointer structure of the tree is still valid and the destruction operation can destroy `R` (c) and `Root` before invalidating the operation descriptor (d).

The only caveat is that an object's destructor may be called more than once. In most cases, this is not a problem, but some idioms that require the destructor to, for instance, track the number of live instances of an object will be more difficult to implement. In our experience using the system, however, this has not been a significant problem.

Reloading NV-heaps It must be possible for one program to load an NV-heap created by another. This presents two challenges. The first is that the NV-heaps may end up mapped into any part of the application's address space. To support this, our system implements NV-to-NV pointers as relative pointers: Instead of holding the actual address (which would change from application to application or execution to execution), the relative pointer holds an offset from the pointer's address to the data it points to. The conversion between relative pointer and virtual addresses requires just a single add instruction.

The second challenge is the virtual table pointers stored in objects with virtual methods. There is no guarantee that the virtual tables or virtual functions will be at the same virtual address across executions. There are (at least) two solutions: The first is to use generation numbers to determine if the virtual table pointer has been update during the current generation (i.e., during the current program's execution), and, if it has not, overwrite it with the correct value (obtained using the dynamic loader). In the common case, this adds an extra integer comparison and branch (to check the generation) to each virtual method call. To call a virtual method, a simple non-virtual stub function can implement the generation-based scheme and call the virtual method. A second approach is to modify the compiler to emit relocatable virtual pointer tables for non-volatile classes and then store the necessary code in the NV-heap. This approach is preferable, but requires changing C++'s application binary interface and type system.

We have implemented the first option.

Recovery To recover from a system failure, the memory allocator performs the following two steps. First, it replays any valid operation descriptors for basic storage allocation and deallocation operations. Then, it replays any reference count updates and reference counting pointer assignments, which may include the recursive destruction process described above. When this is complete, all the reference counts are valid and up-to-date and the recovery process moves on to the transaction system.

4.3.4 Pointers in NV-heaps

The NV-heap library uses operator overloading to implement pointer types for NV-to-NV, V-to-NV, and weak NV-to-NV pointers. Their assignment operators, copy constructors, and cast operators work together with the memory allocator to enforce correct semantics for each pointer type.

The pointers play a key role in preventing the creation of inter-heap NV-to-NV pointers and NV-to-V pointers. NV-to-NV pointers are “wide” and include a pointer to the NV-heap they belong to. Non-volatile objects contain a similar pointer. The assignment operators for the pointer check that the assignment is valid (i.e., that the pointer and the object belong to the same NV-heap).

The smart pointer types also allow NV-heaps to be relocatable. Instead of holding the actual address (which would change from application to application or execution to execution), the pointer holds an offset from the pointer’s address to the data it points to.

Below we describe the implementation of each pointer type.

NV-to-NV pointers NV-heaps support two types of NV-to-NV pointers. Normal NV-to-NV pointers affect reference counts and are the most common type of pointer in the applications we have written. They reside in the NV-heap and point to data in the same NV-heap.

Weak NV-to-NV pointers are similar but they do not affect an object’s

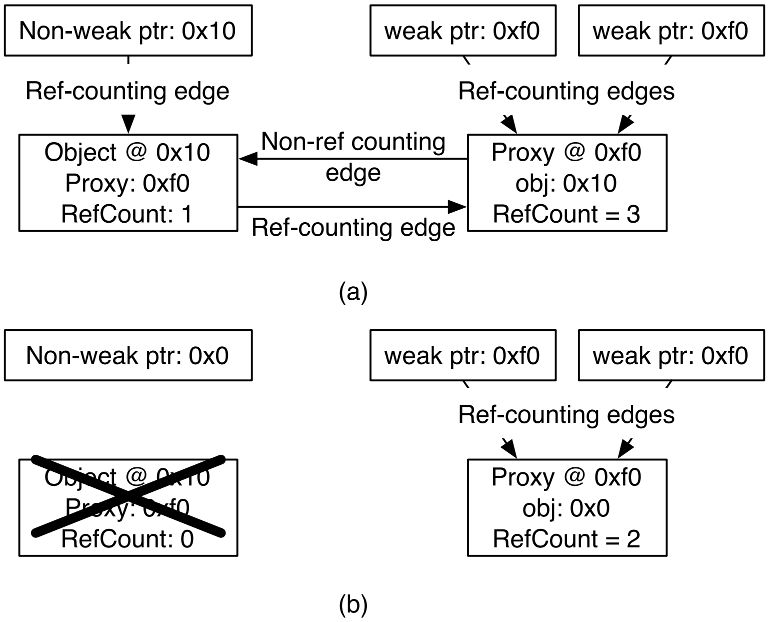


Figure 4.5: **Implementing weak pointers** In (a) two weak pointers and one non-weak pointer all refer to an single object. The weak pointers access the object indirectly via a proxy. When the object is destroyed (b), the weak pointers all instantly become NULL, avoiding unsafe pointers.

reference count. Weak pointers are required to implement cyclic data structures (e.g., doubly-linked lists) without introducing memory leaks. Ensuring that non-null weak NV-to-NV pointers point to valid data requires that when the object they refer to becomes dead (i.e., no more non-weak pointers refer to it), all the weak pointers should atomically become NULL. This may lead to an unexpected NULL pointer dereference, but it cannot result in corrupted data.

There are several options for implementing this behavior. One is to keep a list of all the weak pointers and atomically set them to NULL serially. This would be inefficient if the number of weak pointers were large, and it also requires a variable-sized operation descriptor, which would add considerable complexity.

Instead, we use proxy objects to implement this behavior. Weak NV-to-NV pointers refer indirectly to the object via a proxy that contains a pointer to the actual object. When an object dies, its destructor sets the pointer in its proxy to NULL, instantly nullifying all the weak pointers. The system manages proxy objects with reference counting, similar to regular objects, because they must survive as long as there are weak pointers that refer to them.

Figure 4.5 gives an example. In (a) two weak and one non-weak pointers refer to a single object. In (b) the non-weak pointer becomes NULL, triggering the deallocation of the object. This sets the proxy's non-reference counting pointer to NULL. This makes all weak pointers effectively become NULL as well. The proxy remains intact, since its reference count is non-zero. When the weak references are assigned a different value or are destroyed, the proxy object's reference count will go to zero, and the system will reclaim it.

V-to-NV pointers There are three key requirements for V-to-NV references. First, a V-to-NV reference must be sufficient to keep an object alive. Second, when a program exits and all the V-to-NV references are destroyed, the objects' reference counts must be adjusted accordingly. Third, the system must eventually reclaim any objects that become dead when a program exits and destroys all of the V-to-NV pointers.

To address these issues, we add a second count of V-to-NV references to each object. One attractive solution is to store these counts in volatile memory. This would ensure they were reset correctly on exit, but, in the event of a system or application failure, the V-to-NV reference counts would be lost, making any dead objects now impossible to reclaim.

Our implementation stores the count of volatile references in non-volatile memory and uses the generation number technique we used for non-volatile locks to reset it. To locate orphaned objects, the NV-heap maintains a per-thread list of objects that only have V-to-NV references. On startup, the NV-heap reclaims everything in the list.

4.3.5 Implementing transactions

NV-heaps provide fine-grain consistency through atomic sections that log all updates to the heap in non-volatile memory. Atomic sections for NV-heaps build on previous work developing transactional memory systems for volatile memories [HM93, ST95, HWC+04, BDLM07, SATH+06, HLMS03, HF03]. The NV-heap transaction system is software-only (except for epoch barrier support, and it relies heavily on the transactional memory allocator and reference counting system described in Sections 4.3.3 and 4.3.4. It uses reference-counted objects for all its internal data structures.

Transactions in NV-heaps Our implementation of NV-heaps provides ACID semantics for programs that meet two criteria: First, a program must use the accessor functions to access object fields and it must not use unsafe casts to circumvent the C++ type system. Second, transactions should not access shared *volatile* state, since the NV-heap transaction system does not protect it. We have not found this to be a significant problem in our use of NV-heaps, but we plan to extend the system to cover volatile data as well. Failure to adhere to the second condition only impacts isolation between transactions.

The transaction implementation consists of three key data structures: (i) an NV-heap base class from which all transactional objects inherit, (ii) a set of logs (one per thread) implemented as a linked list of log entries, and (iii) an ownership record table. The object base class provides per-object data including the reference count, object size information, and an object ID.

The ownership record table is the only volatile structure that NV-heaps rely on. Ownership records enforce exclusive write access and detect read-write conflicts during transactions. Each ownership record is protected by a lock, and it stores a pointer to the transaction that currently owns the data, if there is one, and a version number. We use a table of ownership records stored in volatile memory and indexed by the low-order bits of the unique object ID that the system assigns at object creation time. Using a table of ownership records leads to some unnecessary transaction aborts, but it has two key advantages. First, volatile locks are faster than non-volatile locks. Second, using an array indexed by an ID avoids the need for a more complicated map lookup on lock acquisition. In practice, we find that the number of unnecessary aborts to be very low.

NV-heap's log processing is the main difference relative to conventional volatile transactional memory systems. NV-heaps must ensure that if the system fails in the midst of a transaction, the startup recovery code can restore memory to the state it was in before any failed transaction began. This means that the write log must be non-volatile. It also means that, in the case of multiple system failures, it must be safe to re-start the restore procedure at any point. Similarly, in the case of a commit, it must be safe to restart the log deletion at any point. The read log, on the other hand, is volatile.

For each thread accessing the heap, the system maintains a read log and a write log, implemented as linked lists of log entries. When a transaction wants to modify an object, it must first be opened for writing, meaning the system will make a copy of the object so that any changes can be rolled back in the case of an abort or system/application failure. Figure 4.6 contains the pseudo-code for `TXOpenWrite()`. The running transaction must take ownership of the object. If another transaction owns the object already, then a write conflict has occurred and the transaction aborts. After the transaction successfully retries and becomes the owner, the NV-heap copies the object into the log along with a pointer to the original object. To open an object for reading, NV-heaps store a pointer to the original object and its current version number in the log.

The pointer in the log is a reference-counting pointer, just like all other pointers to a transactional object, so creating the copy increments the original object's reference count. Likewise, when we create the copy, the reference counting system also increments the reference counts on any objects referred to by pointers in the object. This is necessary to postpone the deletion of objects until the transaction commits.

When the copy is complete, the application can safely modify the original object. If a system failure occurs or the transaction aborts, the NV-heap rolls back the log entries by restoring the original data from the log, marking the log entry as invalid, and then dropping the pointer to the log entry so that the reference counting system deletes it. Note that is safe to perform the restore multiple times (as might happen if the system crashed during recovery) since, during the re-executions, previously completed assignments will have no effect.

It is also possible that the system will fail during the initial copy. In this

```
void TXOpenWrite(RefCountPtr& p) {
  TransactionPtr trans = getCurrentTransaction();
  int index = hash(p->objID);
  orecTable[index].acquireLock();
  if (orecTable[index].owner == NULL) {
    orecTable[index].owner = trans;
  }
  else if (orecTable[index].owner != trans) {
    orecTable[index].releaseLock();
    TXAbort();
    return;
  }
  orecTable[index].releaseLock();
  NVNew(log->tail->next);
  newEntry = log->tail->next;
  log->tail = newEntry;
  NVNew(newEntry->copy);
  *(newEntry->copy) = *p;
  EpochBarrier();
  newEntry->valid = true;
  EpochBarrier();
}
```

Figure 4.6: **Pseudo-code for opening a transactional object for modification** Before modifications are allowed, the NV-heap copies the object to the log and marks the entry valid.

case, the log entry will be invalid on restart, and dropping the pointer to the entry will trigger its destruction. Here, the danger is in calling a destructor on an object whose constructor has not completed and which might contain garbage pointer values. The memory allocator prevents this by ensuring that the memory it returns is all zeros, guaranteeing that all pointers are initially NULL.

We choose to copy entire objects to the log rather than individual object fields, and this is a common trade-off in transactional memory systems. In NV-heaps, each log operation requires an epoch barrier which has some performance cost. Copying the entire object helps to amortize this cost, and it pays off most when transactions make multiple updates to an object, which is a common feature of the data structures that we studied.

NV-heaps borrow ideas from DSTM [HLMS03], RSTM [MSH$^+$06], DracoSTM [GC07], and McRT-STM [SATH$^+$06]. NV-heaps is a blocking transactional memory implementation that relies on writer locks and read versioning. The system performs eager conflict detection of writes by requiring a transaction to become the owner of an object before modifying it. Read conflicts are detected lazily by validating objects version numbers at commit time. NV-heaps allow direct update of objects in memory by making a copy of each object in the log as it is opened for writing. The system maintains durability by storing undo logs for outstanding transactions in non-volatile memory and rolling back uncommitted changes at restart. The contention management scheme [SS05] backs off and retries in case of conflict. NV-heaps flatten nested transactions into a single transaction.

Transaction abort and crash recovery The processes for aborting a transaction and recovering from a system or application failure are very similar: NV-heaps roll back the transaction by restoring data from the write log into the application's objects, marking log entries invalid, and deleting them as it goes. In the case of a crash, the system first follows the recovery procedure defined in Section 4.3.3 to ensure that the memory allocator is in a consistent state and all reference counts are up-to-date.

An additional concern with crash recovery is that recovery must be restartable in the case of multiple failures. NV-heaps recover from failure by only rolling back valid log entries and using an epoch barrier to ensure that an entry's rollback is durably recorded in non-volatile storage before marking the entry invalid.

4.3.6 Storage and memory overheads

The storage overheads of NV-heaps are small. Each NV-heap contains a control region that holds the root pointer, pointers to the free lists, the operation descriptors, version information, and the current generation number. The storage requirement for the control area is 2 KB plus 1.5 KB per thread for operation descriptors. The NV-heap also uses 0.5 KB of volatile memory.

NV-to-NV and V-to-NV pointers are 128-bits which includes a 64-bit relative pointer and dynamic type information to prevent assignments that would create unsafe pointers. Each object includes 80 bytes of metadata including reference counts, a unique ID, ownership information, a generational lock, and other state. For small objects such as primitive data types and pointers, we provide an array object type to amortize the metadata overhead across many elements.

Supporting transactions requires 80 bytes of per-thread transaction state (e.g., pointers to logs) in addition to storage for the write logs.

4.3.7 Validation

To validate our implementation of the NV-heap allocator and transaction system, we created a set of stress tests that create complex objects and perform concurrent transactions on them. These tests exercise the concurrency and safety of the memory allocation operations, pointer assignments, and the user-defined transactions which run on top of them. We run these tests with up to eight threads for long periods (many hours) and observe no deadlock or data corruption.

To test recovery from failures, we run the tests and kill the program with `SIGKILL` at random intervals. During the recovery process we record which logs and operation descriptors the recovery system processes. Then we perform a consistency check. After killing our test programs thousands of times, we have observed numerous successful recoveries involving each descriptor and log type. In all cases, the recoveries were successful and the consistency checks passed.

4.4 Results

This section presents an evaluation of NV-heaps. We describe the test system and then present experiments that measure basic operation latency. Next, we evaluate its scalability and performance on a range of benchmarks. We examine the overheads that NV-heaps incur to provide strong safety guarantees. Then, we compare NV-heaps to Stasis [SB06] and BerkeleyDB [ora], transactional storage systems that target conventional block devices. Finally, we evaluate performance at the application level by implementing a version of Memcachedb [Chu], a persistent key-value store for dynamic Web applications, using NV-heaps.

4.4.1 System configuration

We present results collected on two-socket, Core 2 Quad (a total of 8 cores) machines running at 2.5 GHz with 64 GB of physical DRAM and 12 MB L2 caches. These machines are equipped with both a conventional 250 GB hard drive and a 32GB Intel Extreme flash-based SSD. We configure the machines with a 32GB RAM disk. We use 24 GB for emulated non-volatile memory and 8 GB for program execution. For the experiments that use disks and the SSD we report "wall clock" timing measurements.

4.4.2 Basic operation performance

Table 4.1 summarizes the basic operation latencies for four different versions of NV-heaps that isolate the various overheads that NV-heaps incur. The first version, NoDur, is a volatile transactional memory system without support for durability. The second version is NV-heaps running on DRAM. The third and fourth versions are NV-heaps running on emulated STTM and PCM, respectively.

The latencies we measure are meant to highlight the overhead of working

Table 4.1: **Basic operation latency for NV-heaps** Support for durability and the increased latency for PCM and STTM both extract a toll in terms of basic operation latencies. Some latencies for "new/delete" are listed as <0.1 because of inconsistencies due to caching effects.

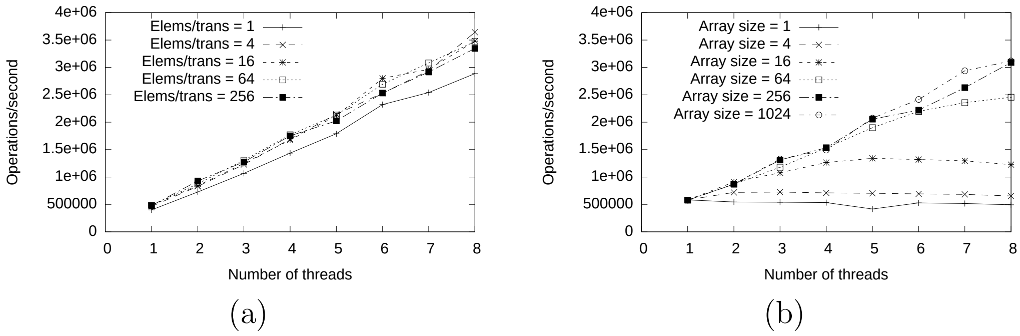
NV-heaps version	NoDur (μs)	DRAM (μs)	STTM (μs)	PCM (μs)
new/delete	<0.1	<0.1	0.75	2.16
V-to-NV ptr	0.08	0.13	0.13	0.13
NV-to-NV ptr	0.13	0.25	0.72	1.68
weak NV-to-NV ptr	0.15	0.25	0.78	1.84
nop tx	0.05	0.05	0.05	0.05
log for read	0.21	0.26	0.26	0.26
log for write	1.00	1.99	5.55	12.67

with persistent objects in NV-heaps. The value for "new/delete" is the time to allocate and deallocate a very small object. The three "ptr" rows give the time to assign to a pointer and then set it to NULL. The "nop tx" is the time to execute an empty transaction. "Log for read" and "Log for write" give the times to log an object before access.

The most expensive operation is logging an object for writing. This operation only occurs once per modified object per transaction and requires an allocation, a copy, one pointer manipulation, and several epoch barriers. The NoDur version of logging an object for writing avoids epoch barriers, so it is able to complete the operation in half the time. The cost of an epoch barrier depends on the size of the epoch, meaning how much data is updated, and whether or not the corresponding cache lines have been written back yet. Our operation descriptors tend to be small, so the cost is often the time to flush a only single cache line.

In contrast, V-to-NV pointer manipulation and read logging are extremely inexpensive, because they do not require durability guarantees or, therefore, epoch barriers. In fact, these operations can occur entirely in the CPU's caches, so the impact of longer memory latencies is minimal.

The PCM and STTM data show the impact that slower underlying memory technology will have on basic operation performance. PCM's longer write latency increases the cost of write logging by 6.4× relative to DRAM. STTM shows a



(a) (b)

Figure 4.7: **Transaction microbenchmark performance.** Graph (a) shows update throughput as we scale the number of elements the transaction modifies, highlighting the overheads of our software in the absence of contention. When multiple thread contend for write access to the same array element, as shown in (b), their execution is serialized. As contention decreases, performance scales more linearly with thread count.

smaller increase — just 2.8×. Compared to NoDur, NV-heaps running in DRAM suffers from an average 1.5× (2× worst case) latency increase due to the added epoch barriers.

Figure 4.7(a) shows how performance scales with transaction size. It measures throughput for updates to an array of one million elements as we vary the number of updates performed in a single transaction from one to 256. The overhead of beginning and completing a transaction is most pronounced for transactions that access only a single element. Increasing the number of elements to four amortizes most of this cost, and beyond that, the improvement is marginal. Overall, the scaling is good: Eight threads provide 7.5× the throughput of a single thread.

Figure 4.7(b) highlights the effect of contention for write access to shared data. We measure write throughput for various array sizes with each transaction accessing only a single element. Under high contention (array sizes of one, four, and 16), we see that throughput is nearly independent of the total number of threads. For larger arrays, contention is minimized and there are fewer aborts which results in an overall throughput that scales with the number of threads, delivering over 6× speedup with 8 threads.

Table 4.2: **NV-heap workloads** We use six workloads of varying complexity to evaluate NV-heaps.

Name	Footprint	Description
SPS	24 GB	Random swaps between entries in an 8 GB array of integers.
SixDegs	8 GB	Concurrently performs two operations: 1) Search for a path of length no more than six between two vertices in a large, scale-free graph 2) modify the graph by inserting and removing edges.
BTree	4 GB	Searches for an integer in a B-tree. Insert it if it is absent, remove it otherwise.
HashTable	8 GB	Searches for an integer in an open-chain hash table. Insert it if it is absent, remove it otherwise.
RBTree	24 GB	Searches for an integer in a 24 GB red-black tree. Insert it if it is absent, remove it otherwise.
SSCA	3 MB	A transactional implementation of SSCA 2.2 [BM05]. It performs several analyses of a large, scale-free graph.

4.4.3 Benchmark performance

Because existing interfaces to non-volatile storage make it difficult to build complex data structures in non-volatile memory, there are no “off the shelf” workloads with which to evaluate our system. Instead, we have written a set of benchmarks from scratch and ported an additional one to use NV-heaps. Table 4.2 describes them.

Figure 4.8 shows the performance of the benchmarks. The numbers inside each bar are the operations (inserts/deletes for BTree, RBTree, and HashTable; path searches for SixDegs; updates or swaps for SPS; iterations through the outer loop of the final phase for SSCA) per second. The graph normalizes performance to NoDur with one thread.

The difference between the NoDur and DRAM bar in each group shows that adding durability to the transaction and memory management systems reduces performance by 40% on average. The cost is largest for SixDegs, because it executes complex transactions and requires frequent epoch barriers. The remaining bars show that the impact of increasing latency of the underlying memory technology is roughly proportional to the change in latency.

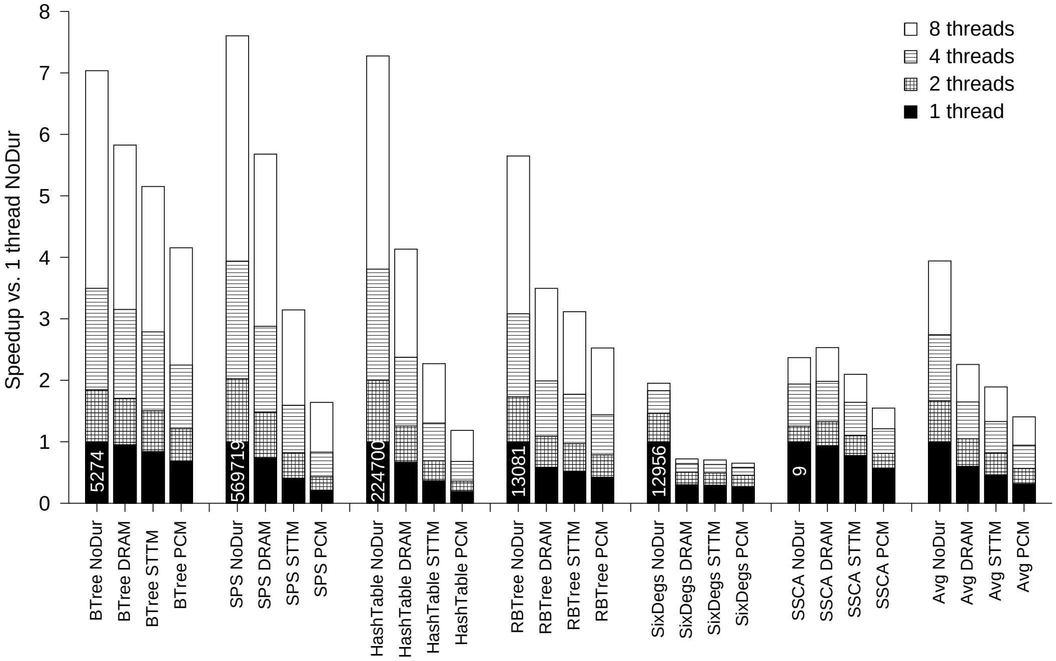


Figure 4.8: **NV-heap performance** Both adding durability and running on non-volatile memory affect NV-heap performance. Numbers in the bars are the throughput for the single-threaded version.

Differences in program behavior lead to varied scaling behavior. SPS, HashTable, and BTree scale very well (7.6×, 7.3×, and 7×, respectively, with 8 threads), while SixDegs and SSCA scale less well due to long transactions and additional conflicts between atomic sections with increasing thread count. We have found that much of this contention is due to the applications rather than NV-heaps. For instance, reducing the search depth in SixDegs from six to one improves scalability significantly.

4.4.4 Safety overhead

To understand the performance overhead of the safety guarantees that NV-heaps offer, we created a version of NV-heaps, similar to Rio Vista [LC97], that provides a bare minimum set of features for building persistent data structures. This version, NV-heap Unsafe, provides checkpoint-based transactions without support for concurrency. It does not provide automatic garbage collection or place any constraints on pointers (e.g., NV-to-V and inter-heap NV-to-NV pointers are allowed).

Figure 4.9 compares the performance of NV-heap Unsafe to our baseline NV-heaps implementation. On average, NV-heap Unsafe performs 8.2× better for these workloads. While the increase in performance is significant, the lack of safety guarantees can be catastrophic in the event of application or system failure. For example, one wrong pointer assignment could corrupt the heap in a way that makes it unusable. To retain the same level of safety as NV-heaps, the programmer must implement many of the same features that NV-heaps provides including logging, locking, and recovery operations. This development cost is high, and we will discuss it in more detail in the next chapter.

The figure also helps quantify the value of avoiding the operating system for common-case accesses. We created a version of NV-heaps, NV-heap Msync, that uses `msync()` in place of epoch barriers to enforce ordering. In this version, a durable update requires a system call and a trip through the Linux IO stack, resulting in large software overheads: NV-heap Msync is 35× slower than normal NV-heaps running on a RAM-disk.

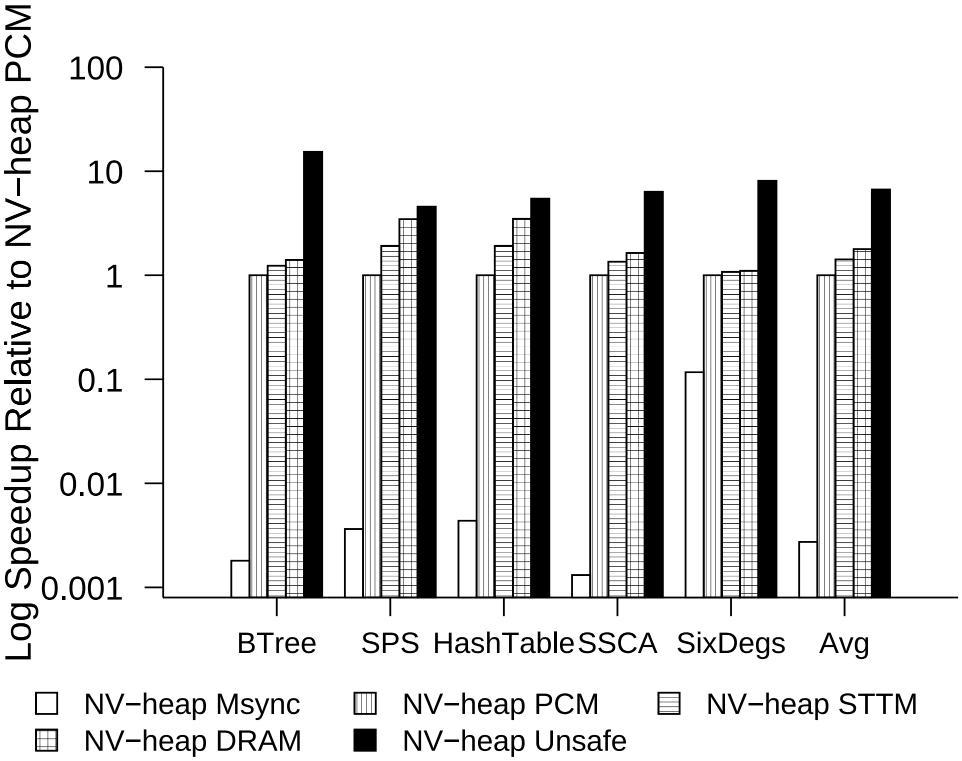


Figure 4.9: **Safety overhead in NV-heaps** Removing safety guarantees improves performance by as much as 8.2× for these workloads, but the resulting system leaves the heap open to data corruption.

4.4.5 Comparison to other systems

This section compares NV-heaps to two other systems that also provide transactional access to persistent state. We compare NV-heaps to Stasis [SB06], a persistent object system that targets disk and BerkeleyDB [SO92], a lightweight database. Both of the systems provide ACID transactions.

NV-heaps, Stasis, and BerkeleyDB have a similar high-level goal — to provide an easy-to-use interface to persistent data with strong consistency guarantees. Stasis and BerkeleyDB, however, target conventional spinning disks rather than byte-addressable storage. This means they must use system calls to force updates to disk and provide durability while NV-heaps take advantage of fast epoch barriers to enforce ordering.

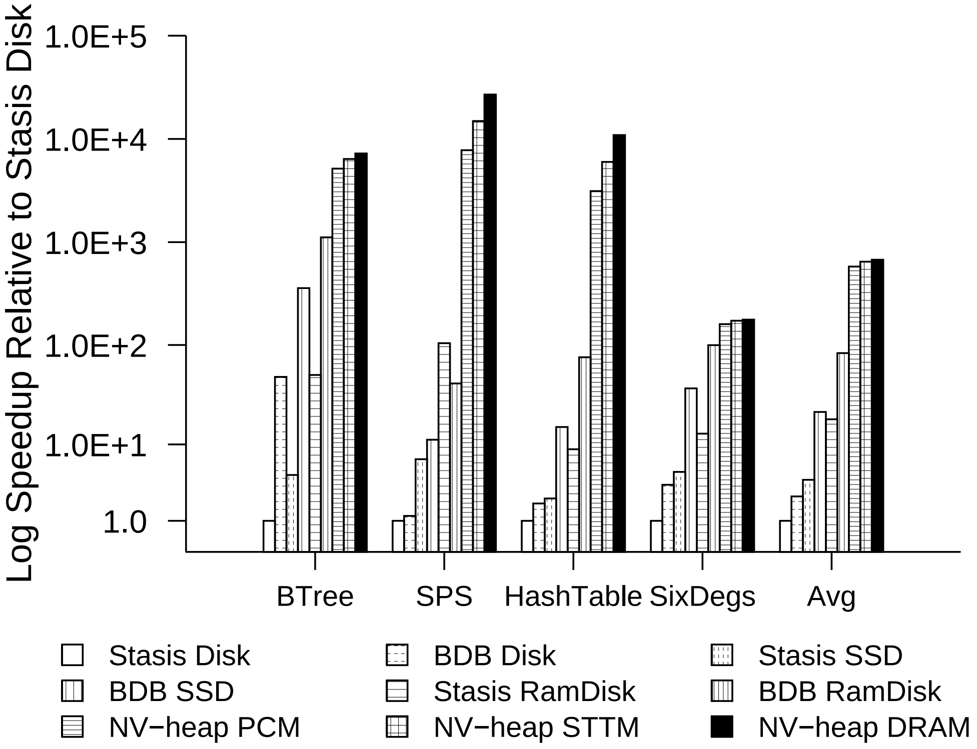


Figure 4.10: **Comparison to other persistent storage systems** NV-heaps outperform Berkeley DB and Stasis by 33 and 244× on average in large part because NV-heaps do not require system calls to provide durability guarantees.

Figure 4.10 compares NV-heap DRAM, NV-heap STTM, and NV-heap PCM to Stasis and BerkeleyDB running on three different hardware configurations: An enterprise hard disk, an Intel Extreme 32 GB SSD, and a RAM-disk. We implemented the data structures in Stasis ourselves. We used BerkeleyDB's built-in BTree and HashTable implementations and implemented SixDegs by hand. The vertical axis is on a log scale. The program runs are for four threads, and we used 4 GB data sets to keep initialization times manageable for the disk-based runs.

The first six bars in each group measure BerkeleyDB's and Stasis' performance on disks, SSDs, and the RAM-disk. The data show that while BerkeleyDB and Stasis benefit from running on fast non-volatile memory (e.g., BerkeleyDB on the RAM-disk is 3.2× faster than on the SSD and 24× than running on disk), the benefits are only a fraction of the raw speedup that non-volatile memories provide compared to SSDs and hard disks.

The next three bars in each group show that NV-heaps do a much better job exploiting that raw performance. NV-heap DRAM is between 2 and 643× faster than BerkeleyDB, and the performance difference for Stasis is between 13 and 814×. Two components contribute to this gap. The first is the `fsync()` and/or `msync()` required for durability on a block device. Removing this overhead by disabling these calls (and sacrificing durability) improves performance by between 2 and 10× for BerkeleyDB. The remaining gap in performance is due to other software overheads.

Comparing NV-heap performance to BerkeleyDB demonstrates that NV-heaps are both flexible and efficient. The NV-heap PCM BTree is 6.7× faster than BerkeleyDB running on the same technology, despite the fact that the BerkeleyDB version is a highly-optimized, specialized implementation. In contrast, the NV-heap implementation uses just the components that NV-heaps provide. We see similar performance for SPS and HashTable. For SixDegs, BerkeleyDB is nearly as fast as the NV-heaps, but this is, in part, because the BerkeleyDB version does not include reference counting. Removing dead nodes from the graph requires a scan of the entire table that stores them.

It is worth noting that our results for comparing BerkeleyDB and Stasis do not match the results in the original Stasis paper [SB06]. We suspect this is due to improvements in BerkeleyDB over the past five years and/or inefficiencies in our use of Stasis. However, assuming results similar to those in the original paper would not significantly alter the above conclusions.

4.4.6 Application-level performance

This section measures the impact of NV-heaps at the application level. We focus on Memcachedb [Chu], a version of Memcached [mem] that provides a persistent key-value store. By default, Memcachedb uses BerkeleyDB to store the key-value pairs and provide persistence.

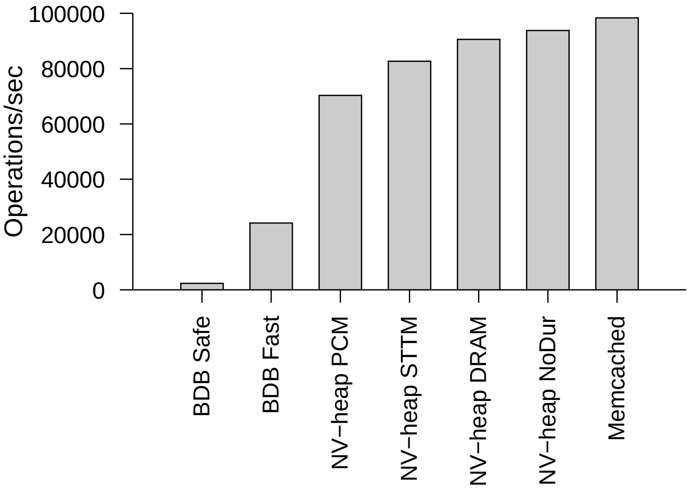


Figure 4.11: **Memcached performance** Using NV-heaps brings performance to within 8% of the original, non-durable Memcached, and the NV-heaps version achieves 39× higher throughput than Memcachedb (BDB Safe) which provides similar safety guarantees.

Our version of Memcachedb uses the NV-heap open-chaining hash table implementation we evaluated in Section 4.4.3 to hold the key-value store. All operations on the key-value store are transactional.

Figure 4.11 shows the throughput of insert/delete operations for the original Memcached application, the Berkeley DB implementation (Memcachedb), and our NV-heap implementation. All tests use 16 byte keys and 512 byte values. The multi-threaded client test program uses the libMemcached [Ake] library, and it runs on the same machine as the server to put maximum pressure on the key-value store. We measure performance of the BerkeleyDB-based version of Memcachedb with (BDB Safe) and without (BDB Fast) synchronous writes. Memcachedb uses the BerkeleyDB hash table implementation and runs on a RAM disk.

Compared to the original Memcached running in DRAM, adding persistence with NV-heaps results in only an 8 to 16% performance penalty depending on the storage technology. When we run NV-heaps without durability (NoDur), performance is within just 5% of Memcached, indicating that the overhead for durability can be low in practice.

The data show that our hash table implementation provides much higher throughput than BerkeleyDB and that the overhead for providing transactional reliability is much lower with NV-heaps. NV-heap DRAM outperforms BDB Safe by up to 39× and NV-heap NoDur outperforms BDB Fast by 3.9×. BerkeleyDB provides many features that our hash table lacks, but for this application those features are not necessary. This highlights one of the advantages of NV-heaps — they allow programmers to provide (and pay the performance penalty for) only the features they need for a particular application.

4.5 Summary

Emerging non-volatile memories will appear on the processor's memory bus, giving programmers direct access to fast storage. While we can build high-performance, persistent data structures in an intuitive and familiar way, we must have strong consistency guarantees in the face of failures or programmer errors. NV-heaps, a system for creating persistent data structures on top of fast, byte addressable, non-volatile memories, provides these guarantees.

NV-heaps prevent several classes of well-known programming errors as well as several new types of errors that arise only in persistent data structures. As a result, NV-heaps allow programmers to implement very fast persistent structures that are robust in the face of system and application failures. The performance cost of the protections that NV-heaps provide is modest, especially compared to the overall gains that non-volatile memories can offer. NV-heaps outperform existing persistent object stores such as BerkeleyDB and Stasis by 32× and 244×, respectively, because they avoid the operating system and lowers software overheads.

While NV-heaps present a new abstraction for storage that provides strong safety guarantees, there are performance overheads due to the features they provide. There are also limitations to what NV-heaps can guarantee as a result of our library-based implementation. For example, programmers can perform arbitrary pointer arithmetic, circumventing our smart pointer types. In the next chapter, we present programming language support and a series of programming models that address these issues. We introduce a core language based on Java that contains the features NV-heaps require, and we describe a novel static dependent type system that enforces the necessary invariants about NV-heaps and references.

Acknowledgments

This chapter contains material from "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories", by Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which appears in *ASPLOS '11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems.* The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2011 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

Chapter 5

Language Support

In Chapter 4, we introduced NV-heaps, a persistent object system designed for fast, non-volatile memories that provides high performance and strong safety guarantees. NV-heaps offer a flexible programming model for storage, but this flexibility creates opportunities for data corruption that do not exist in a system with a well-defined and restricted interface (e.g., block-based file IO in a database). At the same time, the features that NV-heaps make available to ensure safety in the presence of failures and programmer errors (i.e., transactions, garbage collection, safe pointers) can result in significant performance overheads, relative to an unsafe implementation. Ultimately, the usefulness of NV-heaps will depend on the level of the safety guarantees, ease of use, and performance that it provides.

This chapter describes programming language support for NV-heaps in an effort to address these issues. Language support hardens the NV-heaps interface, minimizing the potential for unsafe operations. We present a Java-like language that allows programmers to build persistent data structures, and we describe a new static dependent type system for persistent data, along with the compile-time and run-time support that it requires. The language and type system support several different programming models, outlined below, which offer varying levels of safety, performance, and ease of use.

Layer 1: Base (Section 5.1) The first layer implements a basic model of NV-heaps. It uses a static dependent type system to maintain the key invariants essential for safety and an efficient implementation. Memory management is largely manual, and while very efficient, it does not prevent dangling references. It provides coarse-grained consistency guarantees through checkpointing.

Layer 2: Safe (Section 5.2) The second layer increases safety by adding automatic memory management to Base, while sacrificing some performance.

Layer 3: Tx (Section 5.3) The third layer extends the Safe layer with language support for single-threaded transactions, which allow the programmer to enforce fine-grained consistency in the presence of application or power failures.

Layer 4: C-Tx (Section 5.4) The fourth layer extends Tx with language support for multi-threaded transactions, which allow multiple threads to access an NV-heap while maintaining consistency of persistent data.

The remainder of this chapter is organized as follows. Sections 5.1 through 5.4 describe each programming model and its implementation. Section 5.5 presents a brief survey of related work in language support. Section 5.6 evaluates the performance of the different programming models for NV-heaps. In Section 5.7, we discuss some of the limitations of our current NV-heaps implementation, and in Section 5.8, we present some ideas for future work that would improve the safety, programmability, and performance of our system. Finally, Section 5.9 summarizes the benefits of language support for NV-heaps.

5.1 Layer 1: Base

The first layer, Base, provides the bare minimum facilities for writing programs that can usefully access non-volatile memory. This layer introduces the basic interface to NV-heaps that subsequent layers use.

We present a core language that captures the essence of the programming model NV-heaps expose to the developer. We start by describing the invariants that programs must have to safely and efficiently access NV-heaps (Section 5.1.1). Next, we give an informal overview of the programming model (Section 5.1.2). Then, we present a type system for NV-heaps and show how it ensures the key properties (Section 5.1.3). After that, we describe our implementation of this layer (Section 5.1.4) and discuss the safety and performance implications of programming with Base (Section 5.1.5).

5.1.1 Requirements and Invariants

Base allows developers to safely and efficiently reap the benefits of non-volatile memory by providing a set of facilities that make it easy to use and exploit low-level access to fast storage. Base allows programmers to directly use and manipulate references to locations in an NV-heap. We now describe the requirements to support the Base layer.

Isolation Rather than just expose a monolithic non-volatile store, Base allows programmers to distinguish between different NV-heaps. Explicitly separating NV-heaps makes it easy to *isolate* one store from the others, making it possible to efficiently move or copy a store.

References Effectively isolating NV-heaps from each other and the conventional volatile heap requires us to consider four different types of references (mentioned previously in Section 4.2.3): References within a single NV-heap (*intra-heap* NV-to-NV references), references between two NV-heaps (*inter-heap* NV-to-NV references), references from volatile memory to an NV-heap (V-to-NV references), and references from an NV-heap to volatile memory (NV-to-V references). To enforce isolation, we must ensure that NV-heaps satisfy two critical invariants:

Invariant 1: No NV-to-V references The language must disallow NV-to-V references as they become meaningless once the program ends and would be unsafe the next time the NV-heap is used.

Invariant 2: No inter-heap NV-to-NV references We prohibit inter-store NV-to-NV references as they compromise isolation in several ways. First, inter-heap references complicate garbage collection: It is impossible to tell if a given location in an NV-heap is actually unreachable if a reference in another (potentially unavailable) NV-heap may refer to it. Second, inter-heap references make it difficult to determine (manually, or automatically) whether a given NV-heap can be closed, meaning it is no longer in use by a program.

5.1.2 Programming Model

Base enforces the invariants via a Java-like imperative programming model where each type is *refined* with a *heap-index* that describes the heap in which the associated value is stored. For expressivity, we allow programs to dynamically generate and pass around heap-indices. However, we enforce the key invariants by using a *static dependent* type system. We now describe the components of the language defined by Base (see Figure 5.1).

Heap Indices Base refines each type with a heap index that precisely describes the NV-heap in which the corresponding value resides. A heap index h is either a constant like $\mathtt{h}_{\mathtt{vol}}$ which is a special constant denoting the volatile heap, or a heap variable h_x that describes some heap that will be exactly identified at run-time.

Types We use the heap indices to *refine* each type. Figure 5.1 summarizes the set of types in our language. To simplify the exposition, we use a Java-like model for references, and restrict our types, τ, to be either base values like integers (`Int`) or records composed of a finite set of fields $\mathtt{f}_1 \ldots$ which have the types $\tau_1 \ldots$, respectively. In other words, the programmer does not see explicit pointers. All variables are implicitly references and there is no address-of operation or pointer arithmetic. For example, in our system the following code defines a doubly-linked list of integers:

H	$::=$		**Heap Index**
	$\mid$	$\texttt{c}$	heap constants
	$\mid$	h_x	heap variables
B	$::=$		**Base Types**
	$\mid$	$\texttt{Int}$	integer
	$\mid$	$\texttt{String}$	string
τ	$::=$		**Types**
	$\mid$	B	base
	$\mid$	$\{\tau_1 \texttt{f}_1 \ldots\}$	record
T	$::=$		**Heap Types**
	$\mid$	$\texttt{Heap}(h)$	heap
	$\mid$	$\tau\langle h\rangle$	type-in-heap
e	$::=$		**Expressions**
	$\mid$	x	read-var
	$\mid$	$e.\texttt{f}$	read-fld
	$\mid$	$\texttt{open}(e)$	heap-open
	$\mid$	$\texttt{close}(e)$	heap-close
	$\mid$	$\texttt{checkpoint}(e)$	heap-checkpoint
	$\mid$	$\texttt{heapof}(e)$	heap-of
	$\mid$	$\texttt{null in } e$	null
	$\mid$	$\texttt{new } \tau \texttt{ in } e$	new
	$\mid$	$\texttt{free}(e)$	delete
	$\mid$	$(\tau)\texttt{Root}(e)$	get-root
	$\mid$	$[h \ldots] fn(e \ldots)$	call
s	$::=$		**Statements**
	$\mid$	$\texttt{skip}$	skip
	$\mid$	$T\ x; s$	type-decl
	$\mid$	$s_1; s_2$	sequence
	$\mid$	$\texttt{if } e\ s_1\ s_2$	if-then-else
	$\mid$	$\texttt{while } e\ s$	while loop
	$\mid$	$x{=}e$	write-var
	$\mid$	$e.\texttt{f}{=}e$	write-fld
	$\mid$	$\texttt{Root}(e_1){=}e_2$	set-root
f	$::=$	$\forall h_x \ldots : T\ fn(T_1\ x_1 \ldots)\{$	**Functions**
		$s; \texttt{return } e$	
	$\mid$	$\}$	
p	$::=$	$f_1 \ldots$	**Programs**

Figure 5.1: **Syntax of Types** The BASE programming model provides a Java-like language that allows the programmer to refine a type with a heap-index.

```
type struct DList { Int   data;
                    DList next;
                    DList prev;}
```

Heap-indexed Types A *heap-indexed type* (or heap-type), T, is either (1) $\texttt{Heap}(h)$ denoting an explicit index that identifies a particular NV-heap, or (2) $\tau\langle h\rangle$ where τ is a type and h is the heap-index that describes where values of the type are stored. For example, $\texttt{Heap}(\texttt{h}_{\texttt{vol}})$ is a type whose (singleton) values describe the volatile heap. As another example, $\texttt{DList}\langle h\rangle$ describes a doubly-linked list that is stored in the heap h. We do not refine the individual field of each structure, as our type system ensures that all references stored within a structure (e.g., the `next` and `prev` fields) refer to elements stored in the heap described by the "top-level" index. Note that our type system ensures that heap indices cannot be stored inside records, and hence, cannot be stored inside NV-heaps.

Expressions and Statements The Java-like imperative language of Base includes the expressions and statements summarized in Figure 5.1. The NV-heap is *read from* using variable (x) and field read ($x.\texttt{f}$) expressions, and *written to* using variable ($x{=}e$) and field write ($x.\texttt{f}{=}e$) statements. Other expressions e of our language include variable reads x, constants $\texttt{c}$ (e.g., integer, string constants), primitive operations $e_1\texttt{op}e_2$, and special heap operations that we describe shortly. Other statements s of our language include sequencing $s_1; s_2$, branch statements $\texttt{if}\ e\ s_1\ s_2$, and while-loops $\texttt{while}\ e\ s$. The functions of our language are of the form: $fn(T_1\ x_1 \ldots)\{s; \texttt{return}\ e\}$ where the heap indices appearing in the parameters' types T_1 are (universally quantified) type parameters. A program is a set of functions, with a distinguished function *main*.

Heap Operations Our language contains several new operations that allow the programmer to safely access NV-heaps in a manner that preserves the key invariants. These operations are typed variants of well-known primitives.

$\texttt{open}(e)$ opens the NV-heap identified by the string expression e and returns a heap index corresponding to the newly opened heap. This operation creates or opens a new heap after a program starts.

`close`(e) closes the NV-heap indexed by the value that e that evaluates to. After an NV-heap is closed, all V-to-NV references into it become invalid. We address this danger in Section 5.2.

`checkpoint`(e) atomically commits changes made to the NV-heap indexed by the value that e evaluates to since the last checkpoint or since the NV-heap was opened. Thus, Base provides *checkpoint consistency*: changes to the NV-heap become persistent only when the program asks the NV-heap to take a checkpoint. If the program fails to take a checkpoint, *none* of the program's changes since the last checkpoint are preserved.

`heapof`(e) evaluates to a heap index corresponding to the heap in which the value referred to by the expression e is stored. This operation allows the programmer to determine which heap a value is stored in, which helps in performing operations such as allocating new space in that heap. This operation is included to simplify programming: It relieves the programmer of the burden of passing around heap indices.

`null in` e returns a null reference for the heap indexed by the value that e evaluates to. This operation is used to initialize references in a heap. We associate a heap index with null references so the `heapof`() operation works for all references.

`new` τ `in` e returns a reference to a new τ record in the heap indexed by the value that e evaluates to.

`free`(e) releases the storage occupied by the reference that e evaluates to. We address the danger of multiple deletions and memory leaks in Section 5.2.

(τ)`Root`(e) is a *get root* operation that returns the root reference for the heap that e evaluates to. When the program opens an NV-heap, it does not hold references to any of the data in the heap, and so it cannot access them. The root serves as the unique entry point into the NV-heap, and all "live" data in the heap are reachable from it. If the returned root reference is not of type τ, this operation throws a run-time error. One can think of this as a dynamically checked "downcast" to τ.

`Root`(e_1)=e_2 is a *set root* operation that sets the root for the heap that e_1 evaluates to, to be the reference that e_2 evaluates to. Subsequent get root operations on the same heap must downcast to the same type as e_2, or else they will result in run-time errors.

We now present two code examples to demonstrate the use of heap indices in operations on doubly-linked lists. In the following section, we will describe a set of type checking rules that allow us to statically verify that examples like these maintain our invariants for references.

Example: List Insertion The following code inserts an element `x` at the head of a list `xs`.

```
DList<h> insert(Int x, DList<h> xs){
  DList<h> r;
  r = new DList in heapof(r);
  r.data = x;
  r.next = xs;
  r.prev = null in heapof(r);
  if (xs) {xs.prev = r};
  return r;
}
```

The main difference with the usual insertion procedure is the explicit annotations identifying the heap in which the operations occur. We believe that these will be easy to synthesize using local type inference as done in modern languages like C$^\sharp$and Scala.

Example: Multiplexed-Copy The following code demonstrates a function that takes two lists from (possibly) different heaps, and copies one of them to a third heap `h` (selected by a `flag`).

```
DList<h> copy_mux(DList<hx> xs,
                  DList<hy> ys,
                  Heap h, Int flag){
  Heap h’;
  DList<h’> src;
  DList<h> r, tmp;

  if (flag) {
    h’ = heapof(xs); src = xs;
  } else {
    h’ = heapof(ys); src = ys;
  }
  r = null in h;
  while(src){
    r   = insert(src.data, r);
    src = src.next;
  }
  return r;
}
```

This example shows how explicit heap variables can be used to specify target heaps, as well as indices for references that can refer to more than one heap.

5.1.3 Type System

We use a dependent type system to statically enforce the key invariants needed to use NV-heaps safely and efficiently. This section presents the type checking rules that make up the type system.

Type Environments Our system tracks the prototypes of each function and the types of each variable in-scope using a type environment Γ that maps: (1) function names fn to prototypes that contain the types of the arguments and return value of fn, and (2) variable names x to the declared type of the variable.

Heap Environments Base ensures that at any point where a reference to an NV-heap is read or written the corresponding heap is open. It uses heap environments to determine which heaps are known to be open and to enable a form of heap subtyping described later. A heap environment Δ maps heap variables h_x to

Type Well-Formedness $\boxed{\Gamma, \Delta \vdash T}$

$$\frac{}{\Gamma, \Delta \vdash \mathtt{Heap(c)}}\ \text{[WF-HC]} \qquad \frac{\Delta(h_x) \neq \bot}{\Gamma, \Delta \vdash \mathtt{Heap}(h_x)}\ \text{[WF-HV]}$$

$$\frac{\Gamma, \Delta \vdash \mathtt{Heap}(h)}{\Gamma, \Delta \vdash \tau\langle h \rangle}\ \text{[WF-T]}$$

Function Typing $\boxed{\Gamma \vdash f}$

$$\frac{\Gamma \equiv \Gamma_f \cup \{x_1 : T_1 \ldots\} \quad \Delta \equiv \mathsf{Entry}(T_1 \ldots) \qquad \Gamma, \Delta \vdash s \downarrow \Delta' \quad \Gamma, \Delta' \vdash e : T}{\Gamma_f \vdash \forall h_x : T\ fn(T_1\ x_1 \ldots)\{s; \mathtt{return}\ e\}}\ \text{[T-FUN]}$$

Entry Heap Environment $\boxed{\mathsf{Entry}()}$

$$\begin{array}{ll}
\mathsf{Entry}(\emptyset) & \doteq \lambda h_x.\bot \\
\mathsf{Entry}(\tau\langle h_x\rangle, \ldots) & \doteq \mathsf{Entry}(\ldots)[h_x \mapsto h_x] \\
\mathsf{Entry}(\mathtt{Heap}(h_x), \ldots) & \doteq \mathsf{Entry}(\ldots)[h_x \mapsto h_x] \\
\mathsf{Entry}(T \ldots) & \doteq \mathsf{Entry}(\ldots)
\end{array}$$

Program Typing $\boxed{\vdash f_1 \ldots}$

$$\frac{\Gamma_f \equiv f_1 : \mathsf{Proto}(f_1), \ldots \quad \Gamma_f \vdash f_i \text{ for each } i}{\vdash f_1 \ldots}\ \text{[T-PGM]}$$

Figure 5.2: **Base Type Checking: Well-Formedness, Functions, Programs** In a well-formed type judgement, a heap-indexed type corresponds to a heap that is open.

one of: (1) $\bot$, indicating h_x may be closed, (2) h, indicating that h_x is equal to h and is definitely open, (3) $\top$, indicating that the exact value of h_x is unknown, but it is definitely open. Well-formed type judgements, of the form $\Gamma, \Delta \vdash T$ (shown in Figure 5.2), state that a particular heap-indexed type corresponds to a heap that is known to be open in the given type and heap environments.

Functions and Programs Figure 5.2 shows the typing rules for functions and programs. The function typing judgement of the form $\Gamma \vdash f$ states that the body of a function is well-typed. Formally, the rule [T-FUN] checks that the function implements its prototype, by checking that (1) the function's body s is well-typed in the type environment Γ extended with bindings for the formals and locals and entry heap environment Δ that maps each input heap-parameter h_x to an open heap denoted by itself, and that, (2) in the heap environment Δ' yielded by the body, the return value e has the output type specified in the prototype. Finally, the rule [T-Prog] checks the entire program (a collection of functions), by checking that each function meets its prototype specification, assuming that the other functions meet their specifications.

Expressions Figure 5.3 shows the rules for checking expressions. The expression typing judgement of the form $\Gamma, \Delta \vdash e : T$ states that an expression e has the heap-indexed type T in the type and heap environments Γ and Δ.

- For a read of a heap variable ([T-Rd-Hvar]) our system checks that the variable corresponds to an open heap. The result type is a *singleton* ("self" [OTMW04]) type that the variable refers to.

- For a read of a field ([T-Rd-Fld]) our system checks which heap the reference resides in and looks up the type of the field being read. The result type must reside in the *same heap* as the one e refers to.

- For a new allocation ([T-New]) our system determines which heap h the allocation happens in. The result type is τ indexed by the heap h.

- For a call expression ([T-Call]) our system substitutes the (implicit) actual heap-indices $h \ldots$ for the formal heap-indices $h_x \ldots$ of the callee's prototype and checks that each actual heap-index has the type of the corresponding formal after substitution. The result type is the callee's output type after substitution.

Statements Figure 5.4 shows the rules for checking statements. The statement typing judgement of the form $\Gamma, \Delta \vdash s \downarrow \Delta'$ states that a statement s is well-typed in the type and heap environments Γ and Δ, and, upon finishing, yields the heap environment Δ'.

Expression Typing $\boxed{\Gamma, \Delta \vdash e : T}$

$$\frac{\Gamma(x) = \mathtt{Heap}(\cdot) \quad \Delta(x) \neq \bot}{\Gamma, \Delta \vdash x : \mathtt{Heap}(x)}\ [\text{T-}\textsc{Rd-Hvar}]$$

$$\frac{\Gamma(x) = T}{\Gamma, \Delta \vdash x : T}\ [\text{T-}\textsc{Rd-Var}] \qquad \frac{\Gamma, \Delta \vdash e : \tau\langle h\rangle \quad \tau \equiv \{\tau_{\mathtt{f}}\ \mathtt{f} \ldots\}}{\Gamma, \Delta \vdash e.\mathtt{f} : \tau_{\mathtt{f}}\langle h\rangle}\ [\text{T-}\textsc{Rd-Fld}]$$

$$\frac{\Gamma, \Delta \vdash e : \mathtt{Heap}(h)}{\Gamma, \Delta \vdash \mathtt{null\ in}\ e : \tau\langle h\rangle}\ [\text{T-}\textsc{Null}]$$

$$\frac{\Gamma, \Delta \vdash e : \mathtt{Heap}(h)}{\Gamma, \Delta \vdash \mathtt{new}\ \tau\ \mathtt{in}\ e : \tau\langle h\rangle}\ [\text{T-}\textsc{New}]$$

$$\frac{\Gamma, \Delta \vdash e : \mathtt{String}}{\Gamma, \Delta \vdash \mathtt{open}(e) : \mathtt{Heap}(h)}\ [\text{T-}\textsc{Open}]$$

$$\frac{\Gamma, \Delta \vdash e : \mathtt{Heap}(h)}{\Gamma, \Delta \vdash (\tau)\mathtt{Root}(e) : \tau\langle h\rangle}\ [\text{T-}\textsc{GetRoot}]$$

$$\frac{\Gamma, \Delta \vdash e : \mathtt{Heap}(h)}{\Gamma, \Delta \vdash \mathtt{heapof}(e) : \mathtt{Heap}(h)}\ [\text{T-}\textsc{Heapof}]$$

$$\frac{\Gamma, \Delta \vdash e_1 : \mathtt{Heap}(h_1) \qquad \Gamma, \Delta \vdash e_2 : \tau\langle h_2\rangle \qquad \Delta \vdash h_1 = h_2}{\Gamma, \Delta \vdash \mathtt{Root}(e_1) {=} e_2 : \mathtt{Int}}\ [\text{T-}\textsc{Set-Root}]$$

$$\frac{\Gamma, \Delta \vdash e : \tau\langle h\rangle}{\Gamma, \Delta \vdash \mathtt{free}(e) \downarrow \Delta}\ [\text{T-}\textsc{Del}] \qquad \frac{\Gamma, \Delta \vdash e : \mathtt{Heap}(h)}{\Gamma, \Delta \vdash \mathtt{close}(e) : \mathtt{Int}}\ [\text{T-}\textsc{Close}]$$

$$\frac{\Gamma, \Delta \vdash e : \mathtt{Heap}(h)}{\Gamma, \Delta \vdash \mathtt{checkpoint}(e) : \mathtt{Int}}\ [\text{T-}\textsc{Checkpoint}]$$

$$\frac{\Gamma(fn) = \forall h_x : T\ fn(T_1\ x_1 \ldots) \qquad \Gamma, \Delta \vdash e_i : [h \ldots / h_x \ldots]T_i \text{ for each } e_i}{\Gamma, \Delta \vdash [h \ldots]fn(e_1 \ldots) : [h \ldots / h_x \ldots]T}\ [\text{T-}\textsc{Call}]$$

Figure 5.3: **Base Type Checking: Expressions** A heap variable used in a expression should refer to an open heap.

Statement Typing $\boxed{\Gamma, \Delta \vdash s \downarrow \Delta'}$

$$\frac{}{\Gamma, \Delta \vdash \texttt{skip} \downarrow \Delta}\ [\text{T-SKIP}]$$

$$\frac{x \notin \mathsf{Dom}(\Gamma) \quad \Gamma; xT, \Delta \vdash s \downarrow \Delta'}{\Gamma, \Delta \vdash T\ x; s \downarrow \Delta'}\ [\text{T-DECL}]$$

$$\frac{\Gamma, \Delta \vdash s \downarrow \Delta_1 \quad \Gamma, \Delta_1 \vdash s' \downarrow \Delta'}{\Gamma, \Delta \vdash s_1; s_2 \downarrow \Delta'}\ [\text{T-SEQ}]$$

$$\frac{\Gamma, \Delta \vdash e : T \quad \Gamma, \Delta \vdash s_1 \downarrow \Delta_1 \quad \Gamma, \Delta \vdash s_2 \downarrow \Delta_2}{\Gamma, \Delta \vdash \texttt{if}\ e\ s_1\ s_2 \downarrow \mathsf{Join}(\Delta_1, \Delta_2)}\ [\text{T-IF}]$$

$$\frac{\Gamma, \Delta \vdash e : T \quad \Gamma, \Delta \vdash s \downarrow \Delta'}{\Gamma, \Delta \vdash \texttt{while}\ e\ s \downarrow \Delta}\ [\text{T-WHILE}]$$

$$\frac{\Gamma(x) = \texttt{Heap}(\cdot) \quad \Delta(x) = \bot \quad \Gamma, \Delta \vdash e : \texttt{Heap}(h)}{\Gamma, \Delta \vdash x{=}e \downarrow \Delta[x \mapsto h]}\ [\text{T-WR-HVAR}]$$

$$\frac{\Gamma(x) = B\langle\cdot\rangle \quad \Gamma, \Delta \vdash e : B\langle\cdot\rangle}{\Gamma, \Delta \vdash x{=}e \downarrow \Delta}\ [\text{T-WR-VAR-B}]$$

$$\frac{\Gamma(x) = \tau\langle h_x\rangle \quad \Gamma, \Delta \vdash e : \tau\langle h\rangle \quad \Delta \vdash h = h_x}{\Gamma, \Delta \vdash x{=}e \downarrow \Delta}\ [\text{T-WR-VAR}]$$

$$\frac{\Gamma, \Delta \vdash e_1 : B\langle\cdot\rangle \quad \Gamma, \Delta \vdash e_2 : B\langle\cdot\rangle}{\Gamma, \Delta \vdash e_1.\texttt{f}{=}e_2 \downarrow \Delta}\ [\text{T-WR-FLD-B}]$$

$$\frac{\Gamma, \Delta \vdash e_1 : \tau\langle h_1\rangle \quad \tau \equiv \{\tau_{\texttt{f}}\ \texttt{f} \ldots\} \qquad \Gamma, \Delta \vdash e_2 : \tau\langle h_2\rangle \quad \Delta \vdash h_2 = h_1}{\Gamma, \Delta \vdash e_1.\texttt{f}{=}e_2 \downarrow \Delta}\ [\text{T-WR-FLD}]$$

Figure 5.4: **Base Type Checking: Statements** Type checking a statement with heap-refined types requires checking that the heaps are open and that source and target heaps are equivalent.

- For a write of a heap variable ([T-Wr-Hvar]), our system extends the heap environment with a binding for the heap variable. If the expression being written is well-typed, then it corresponds to an open heap, thus maintaining the invariant that all non-$\bot$ bindings in the heap environment refer to open heaps.

- For a base type write through a variable or field ([T-Wr-Var-B], [T-Wr-Fld-B]), our system ignores the source and target heap indices as the base value is copied. For a non-base write through a variable or field ([T-Wr-Var], [T-Wr-Fld]) our system checks that the source and target heaps are equivalent under the heap environment.

- For branch statements ([T-If]) our system checks the then- and else- statements. The heap environment yielded by the branch is the *join* of the heap environments at the end of the two branches, which describes the heap environment that is guaranteed to hold regardless of which branch was taken. Formally, the join is described as

$$\mathsf{Join}(\Delta_1, \Delta_2) \doteq \lambda h_x.\mathsf{Join}(\Delta_1(h_x), \Delta_2(h_x))$$

$$\mathsf{Join}(h_1, h_2) \doteq \begin{cases} h_1 & \text{if } h_1 = h_2 \\ \bot & \text{if } h_1 = \bot \ \text{ or } h_2 = \bot \\ \top & \text{otherwise} \end{cases}$$

Subtyping via Equalities We use the equalities tracked by the heap environment to add flexibility to our type system. The equalities allow us to remove the restriction that each variable must refer to exactly one heap. This is illustrated by the `copy_mux` example above, in which the variable `h'` is used to refer to the heap that `src` refers to. The assignments to `src` typecheck since the heaps that the left- and right- hand sides of the assignments refer to are equal under their respective heap environments. Thus, the equalities tracked by the heap environments enable a form of subtyping without imposing the nested (LIFO) lifetimes required by previous systems [HMGJ04].

Static Guarantees Our type system guarantees that (1) heap indices are not stored in non-volatile storage and (2) all references stored in a heap refer to exactly the same heap, meaning that there are no inter-heap NV references. In other words, we get a compile-time guarantee that the key invariants are always maintained. However, the explicit `free`() and `close`() operations mean that two kinds of dangling references may exist: A reference to an open NV-heap may refer to deallocated storage, or a reference may refer to a location in a closed NV-heap.

5.1.4 Implementation

We have implemented the core functionality of Base as a C++ class library, described in detail in Section 4.3. The implementation stores the contents of NV-heaps as files in a conventional file system and uses `mmap()` to map them into the application's address space. It provides "smart pointers" that implement NV-to-NV and V-to-NV pointers, a non-volatile memory allocator, and a checkpointing facility. Eventually, NV-heaps should be implemented in a compiler that supports the core language and type system directly.

Relocation A key implementation detail is how to make the NV-heap relocatable. It must be possible for one program to load an NV-heap created by another. This means that the NV-heaps may end up mapped into any part of the application's address space. As discussed in Section 4.3.4, we implement NV-to-NV pointers as relative pointers: Instead of holding the actual address (which could change when an application re-opened an NV-heap), the relative pointer holds the offset from the pointer's address to the data it points to. The conversion between relative pointer and virtual addresses requires just a single add instruction.

Checkpointing To implement checkpointing, we pass the `MAP_PRIVATE` option to `mmap()` which prevents changes from propagating to permanent storage until a checkpoint is requested. To take a checkpoint, the system creates a temporary file, writes out the contents of the NV-heap and then replaces the existing file with the copy. Since the `rename` system call is atomic, the persistent changes to the NV-heap are atomic as well.

5.1.5 Discussion

The Base layer enforces the key invariants that NV-heaps require, and it offers the best possible performance for read and write operations (see Section 5.6). Accessing non-volatile data requires (at most) an extra arithmetic instruction to convert a relative NV-to-NV pointer before dereferencing it. However, the layer suffers from two drawbacks.

First, the explicit `free`() and `close`() operations compromise the safety of pointers, leaving open the door to dangling pointers, multiple-frees, and memory leaks which are especially pernicious in an NV-heap as there is no way to ever reclaim the leaked storage.

Second, the consistency mechanism is not scalable, since each checkpoint requires copying the entire NV-heap. The consistency mechanism is also coarse-grained, since the program must take an explicit checkpoint to ensure durability. Fine-grained consistency guarantees are useful in many applications as they provide a more efficient way to maintain key invariants for important data.

5.2 Layer 2: Safe

The second layer, Safe, builds upon the first to ensure the safety of all operations that involve references. To do so, the layer Safe uses run-time mechanisms for safely deallocating storage and closing NV-heaps, as described next. Thus, the language for Safe is a strict subset of that from Base.

5.2.1 Safe Closing

Our current implementation retains the explicit close operation, and throws an exception on accesses to closed NV-heaps. The application must catch and handle the exception in order to recover.

In the longer term, we would like to remove the `close`() operation altogether and replace it with an automatic means of determining when there are no references from (volatile) program variables to an NV-heap. To implement this type of garbage collection, we need simply count, for each open NV-heap, the number of live V-to-NV references that point into the heap. This count is incremented whenever a new reference (or heap variable, h) is created (either by `open`(), `new in` , `Root` or copying), and decremented when the reference is destroyed or goes out of scope. An NV-heap can be closed automatically when the count goes to zero. This is similar to the reference counting system for region-based memory allocators described in [GA01].

5.2.2 Automatic Deallocation

The SAFE layer has no explicit `free`() operation, so it requires a memory management scheme that automatically reclaims any data that are unreachable via any combination of V-to-NV and NV-to-NV references. The SAFE layer uses the reference counting scheme for automatic memory management described in Section 4.3.3, including special support for cyclic data structures. We could not use traditional garbage collection techniques which scan the entire NV-heap to reclaim unused space because they would violate our scalability requirement.

To track reference-counts for NV-to-NV references, we modify our implementation of `new in` to allocate extra space for the reference count (and a few other pieces of metadata, described below) and update the count when the program modifies an NV-to-NV pointer to the allocated storage. Keeping reference counts for V-to-NV references is similar but presents a scalability problem as programs can take checkpoints while V-to-NV references exist that point into the NV-heap. If the program later re-opens the NV-heap in that state, we must update the reference counts to reflect the disappearance of the V-to-NV references.

As described in Section 4.3.4, our implementation maintains *separate* V-to-NV and NV-to-NV reference counts for each piece of allocated storage, thereby *reducing* the problem to that of first, resetting the V-to-NV counts when an NV-heap opens, and second, reclaiming storage that was only reachable from volatile references.

Resetting volatile counts Each NV-heap maintains a *generation number* and increments this number each time it opens. Each allocated region of memory within the NV-heap keeps a (persistent) copy of the generation number for which its V-to-NV reference count is valid. When the reference counting system creates a V-to-NV reference to a block, it first checks the generation number. If it is stale, it resets the count to zero and updates the generation number before incrementing the count.

Reclaiming memory reachable from volatile references When an object's NV-to-NV reference count goes to zero, but its V-to-NV reference is non-zero, the system adds the object to a non-volatile list of memory blocks that will be *dead-on-open*. Creating an NV-to-NV reference to a block removes the block from the list. Destroying the last V-to-NV reference to the block causes its removal and reclamation. When an NV-heap opens, it reclaims any blocks in the dead-on-open list. This approach meets our scalability criteria, since the length of the dead-on-open list (and the time needed to reclaim it) is bounded by the amount of volatile memory in the system, since at least one V-to-NV reference exists to each block in the list.

Weak References Since we use reference counting to identify unreachable data, cyclic structures require special care. To support them we provide a type of *weak* NV-to-NV reference that does not affect reference counts (see Section 4.3.4). Weak references themselves present a challenge, since the data they refer to can evaporate at any moment, if the last non-weak reference to the data disappears. We avoid this problem by requiring that when the system reclaims a piece of data, all weak references to the data instantly become NULL. This prevents the program from ever dereferencing a weak-reference whose storage has been reclaimed, but does not prevent it from dereferencing an (unexpectedly) NULL weak reference. While the former could lead to data corruption and must be avoided at all costs, the latter will lead to a thrown exception which, if not caught, will cause a program crash. In any case, the consistency model guarantees that the NV-heap will remain consistent.

To support instantly setting all weak references to a deallocated piece of storage to NULL, we implement weak references using proxy objects. Instead of pointing to the data itself, the weak references point to the data's proxy. The proxy contains a reference to the block, but this reference does not affect the block's reference count. The block, in turn, holds a reference to the proxy. When the block's reference count goes to zero, it sets the proxy's reference to NULL, instantly causing all the weak references to become NULL. The proxy itself is reference counted and persists until all the weak references that point to it have been destroyed.

It is important to point out that weak references guard against cycles only if the programmer uses them correctly. This means that, while they provide some guarantee storage can eventually be reclaimed, that guarantee is not as strong as it could be if programming language and run-time system could enforce it. In Section 5.8, we discuss garbage collection alternatives that could make this guarantee stronger.

5.2.3 Discussion

The changes described for Safe make references safe and remove the problems associated with explicit memory deallocation. However, V-to-NV and NV-to-NV reference counting add overhead to every reference assignment, and the proxies add overhead to dereferencing and creating weak references. We will examine the impact of these overheads in Section 5.6.

In our experience, the Safe layer makes it is easy to develop complex, reference-based data structures and be confident that they are correct and will not leak memory. However, taking checkpoints is expensive and requires that all accesses to the NV-heap cease temporarily to ensure that the checkpoint is internally consistent. This suspension of activity will hurt performance significantly when multiple threads operate on a heap concurrently. The next section removes these shortcomings by providing a fine-grained consistency model via atomic sections.

5.3 Layer 3: Tx

The most natural way to provide fine-grained consistency guarantees for NV-heaps is adding support in the language for atomic sections implemented with transactional memory. With the addition of atomic sections, we call this programming model the Tx layer.

Transactional memory has received a great deal of attention in recent years as an alternative lock-based synchronization [HM93, ST95, HWC+04, BDLM07, SATH+06, HLMS03, HF03]. For non-volatile memory, however, transactions are useful even in the absence of concurrency, because they provide atomicity and durability guarantees. Transactions in Tx support only a single thread. The last layer we present remedies this shortcoming.

We now briefly describe the application interface and then describe the transaction system's internal implementation.

5.3.1 Language support for fine-grain consistency

At the language level, the changes required are small. The Tx layer adds an atomic keyword (similar to [HF03]) that requires that the operations within a region of code are atomic and durable. IO operations are not allowed within atomic sections, and the transaction layer flattens nested atomic regions into a single region.

The Tx layer makes another small but important change to the language: It prevents memory leaks in the face of crashes. Specifically, this means the `new in` operation must be atomic and durable to enforce the following invariant: At any time, every storage cell is either the property of the allocator (i.e., it is not allocated) or of the program (i.e., it is allocated). In either case, the owner holds a reference to the region. Regions of non-volatile storage must move between these two states atomically and the changes must be permanent, regardless of unexpected program or system failures.

5.3.2 Implementing atomic, durable memory allocation

Since the transaction system will, itself, need to allocate memory, we cannot rely on it to provide atomic and durable memory management. Our implementation uses special-purpose logs called *operation descriptors* to ensure atomicity and durability. The descriptors record all changes needed to perform the allocation (or deallocation) in non-volatile memory. To perform an operation, the NV-heap fills out an operation descriptor, marks it valid, and then applies the operation. Operation descriptors are idempotent by design, so in the case of failure, the NV-heap can replay the descriptor to complete the operation the next time it opens. Once the operation is complete, the NV-heap marks the descriptor as invalid.

In addition to allocation and deallocation, modifications to references and the corresponding reference count updates must be atomic and durable operations. The system uses operation descriptors for these operations as well.

Caching effects in the memory hierarchy may reorder and arbitrarily delay writes to non-volatile memory cells, preventing them from actually reaching non-volatile storage. The system could reorder the writes that mark the operation descriptors as valid relative to the writes that fill in the operation descriptor, making it appear that an incomplete descriptor was, in fact, valid.

To avoid this, NV-heaps provides a mechanism to ensure that updates have reached non-volatile storage and are, therefore, permanent. We assume that hardware provides atomic 8-byte writes and an epoch barrier operation, as described in [CNF$^+$09], that blocks until a set of previous updates to non-volatile storage locations are permanent. An epoch barrier is similar to the `fsync()` system call, in that it allows the program to enforce an ordering of updates to durable storage. Inserting a barrier before and after setting the valid bit ensures that only valid operation descriptors are marked valid. Using epoch barriers to provide fine-grained consistency guarantees, as we do for operation descriptors, is critical to application safety. In Section 5.8, we discuss language support that could help ensure the proper placement of these barriers in user code.

5.3.3 Implementing single-threaded transactions for non-volatile memory

The transaction layer must ensure that if the system fails in the midst of a transaction, the startup recovery code can restore memory to the state it was in before the failed transaction began. This means that the log that contains the backup copy of the modified data must be stored in the NV-heap. It also means that, in the case of multiple system failures, it must be safe to re-start the restore procedure at any point. Similarly, in the case of a commit, it must be safe to restart the log deletion at any point.

The log is a linked list of log entries, and it is created and managed by the run-time system. Before modifying an object inside a transaction, the object must be opened for writing, which means the system copies the object into the log along with a reference to the original object. For reading, there is no required opening operation prior to access.

The reference to the object that is saved in the log is an NV-to-NV reference, so creating the copy increments the object's reference count. Likewise, when we create the copy of the object, the reference counting system also increments the reference counts on any objects referred to by references in the object. This postpones the deletion of objects until the transaction commits.

After the copy is complete, the application can safely modify the original region. If a system failure occurs or the transaction aborts, the system rolls back each log entry by restoring the original data from the log, marking the log entry as invalid, and then dropping the reference to the log entry so that the reference counting system deletes it. Note that it is safe to perform the restore multiple times (as might happen if the system crashed during recovery) since the recovery process is just a copy operation.

It is also possible that the system will fail during the initial copy. In this case, the log entry will be invalid on restart, and dropping the reference to the entry will trigger its destruction. Here, the danger is in calling a destructor on an object whose constructor has not completed and which might contain garbage reference values. The memory allocator prevents this by ensuring that the memory it returns is all zeros, guaranteeing that all references are initially NULL.

5.3.4 Discussion

Atomic sections provide a more natural and composable mechanism for implementing atomic and durable operations. However, the cost of logging copies of all data before performing the operation adds significant overhead as we will see in Section 5.6. The next layer allows programs to reclaim that lost performance by providing support for concurrent transactions.

5.4 Layer 4: C-Tx

The layers described so far do not provide support for safe, concurrent access to an NV-heap by multiple threads. Adding support for concurrency on top of the Tx layer requires the addition of threading to our core language. It also requires the system to provide scalable locks for non-volatile objects and significant changes to the transaction system.

5.4.1 Locks for protecting non-volatile data

Providing concurrent access to NV-heaps requires locks to protect the reference counts for each object. Keeping these locks in volatile memory is attractive, since access to them would be faster and all locks would be automatically released if the system or application crashed. However, volatile locks violate our scalability requirements since the number of locks required grows with the size of the NV-heap.

Storing locks in non-volatile memory resolves that storage scalability concern, but raises a performance scalability problem: Each time an NV-heap opens, it would need to scan all the locks in the NV-heap and unlock them. Non-volatile locks use the generation numbers described in Section 5.2 to address this problem: An *generation-based lock* is an integer and, if the integer is equal to the NV-heap's current generation number, a thread holds it. Otherwise, it is available.

Incrementing the NV-heap's generation (as happens when the NV-heap opens) instantly releases all the locks in the NV-heap.

5.4.2 Concurrent memory allocation

Supporting concurrency requires changes to the memory allocator and reference counting system provided by the Base and Safe layers. The concurrent allocator provides a private free list for each thread to reduce contention. The allocator also contains a global free list and periodically moves free space from the per-thread lists into the global list. Each thread also has a private set of operation descriptors for performing allocations and manipulating reference-counting pointers. This is described in detail in Section 4.3.3.

5.4.3 Concurrent transactions

Adding support for concurrency to our transaction system adds significant complexity. It must detect conflicting accesses to regions of memory by different threads and provide the ability to abort transactions when a conflict occurs.

To track conflicts, the transactional memory system uses a second linked list to hold the read log. When a thread opens an object for reading, it copies a reference to the object into the read log along with a version number. The transactional memory system maintains a separate log for each thread.

The transactional memory system does not use generation-based locks to protect data in storage. Instead, it uses a volatile table of ownership records. Ownership records [HF03] enforce exclusive write access and detect read-write conflicts during transactions. Each ownership record is protected by a reader/writer lock. See Section 4.3.5 for a more detailed discussion of the implementation.

The multi-threaded transactional memory system is a blocking implementation that relies on writer locks and read versioning. It performs eager conflict detection of writes by requiring a transaction to acquire ownership of the object before modifying it. It detects read conflicts by validating object version numbers at access and at commit. We use a contention management scheme [SS05] that exponentially backs off and retries in case of conflict.

To implement abort, the transactional memory system must roll back the current transaction. This process is similar to recovering from a failure in the Tx layer. The transactional memory system rolls back the transaction by restoring data from the write log into the application's memory, marking log entries invalid, and deleting them as it goes.

Our implementation of the C-Tx layer provides nearly ACID (atomic, consistent, isolated, and durable) semantics for its transactions. Its only limitation is with respect to isolation: Since the transactional memory system is a software-only scheme based on C++, it is not possible to provide complete isolation. NV-heaps do, however, require the use of "getters" and "setters" for field access and these perform run-time checks to enforce isolation in well-behaved programs. While this requirement places the burden on the programmer, we could provide compiler support that would automatically generate and instantiate these methods.

5.5 Related Work

In this section, we discuss language support for NV-heaps in the context of previous work in programming languages, compilers, and run-time systems. Specifically, we focus on the techniques we use to provide safety in each programming model (e.g., dependent types, garbage collection, transactional memory), and we compare them to existing methods.

Our concept of heap-indices, introduced in Section 5.1 for the Base layer, is related to the notion of *region-based* memory management [TT97, GA01, HMGJ04], where instead of allocating and freeing individual pointers, the programmer first creates coarse-grained *regions* (also known as zones or arenas). Individual cells are allocated within regions, and the programmer must pass the region in as an explicit parameter to the allocator. The key benefits include the fact that the entire region, including all the cells inside it, is freed at once and type-based mechanisms ensure that the program never accesses a pointer that has already been freed. Due to the persistence of storage, the invariants required to efficiently use NV-heaps are different from those that must hold for regions, which can, for example, have inter-region references. First, persistence places a high premium on isolation, which is delivered via the key isolation invariants, which allow us to avoid complex, inter-heap garbage collection[ML97]. Second, persistence implies that NV-heaps will typically not be used in a lexically scoped fashion. Consequently, we cannot use nested lifetimes as a basis for a subtyping relation on non-volatile references.

Our static dependent type system is related to several dependent and refinement type systems that have been developed to enforce domain specific properties [XP99, BBF$^+$08]. Of these, the most closely related is the idea of place types [CSSB08] which decorate references in the (distributed) X10 language with an index denoting where the value is stored, with the goal of ensuring that all computations are carried out on locally stored data. The invariants required by NV-heaps are different, and hence we have a "path"-sensitive type system that tracks equalities of heap variables (in the heap environment), allowing for a flexible form of subtyping.

The Safe layer relies on a reference counting [Col60] scheme for garbage collection because it is deterministic and scalable. NV-heaps use weak references for cyclic data structures, relying on the programmer to identify possible back pointers. Alternatively, there are several existing algorithms to automatically collect cycles [Lin92, MWL90, BR01, PBK$^+$07] which could replace our existing approach. In addition to reference counting, there are other garbage collection schemes [KW93, ONG93, CKWZ96, MMH96] for non-volatile storage, but they target an architecture based on disk. Also, these schemes split the heap into partitions and require complex techniques to track inter-heap references efficiently [ML97]. Our Safe layer ensures that an NV-heap is automatically and safely closed when the reference counts of all V-to-NV references are zero. This is similar to the reference counting system used for region-based memory allocators [GA01], where regions are freed when there are no remaining references to them.

Tx and C-Tx owe much of their heritage to the volatile transactional memory [HM93] systems that provide an alternative to locks for managing concurrent access to volatile memory. Recent work has focused on utilizing conventional coherence and consistency protocols and overcoming data size restrictions [HWC+04, AAK+05, RHL05, BDLM07]. Given the expense of hardware support, Shavit and Touitou proposed a software transactional memory system which provided weak isolation and explicit statically-sized transactions [ST95]. The operation descriptors that the NV-heaps allocator and reference-counting system use are similar. Dynamic software transactional memory (DSTM) provided strong isolation and overcame the static limitation to allow dynamic data structures such as lists and trees [HLMS03]. The object-based transactional memory model in NV-heaps is similar to that of DSTM and others [MSH+06, GC07]. Parts of our internal algorithms for read versioning, write locking, and undo logging were inspired by McRT-STM, a C++ library-based multi-core runtime system that supports software transactional memory [SATH+06].

Next, we evaluate the performance and safety of the programming models presented in this chapter.

5.6 Results

This section describes our evaluation of the language layers for NV-heaps. We present experiments that measure basic operation latency for each of the layers, and we examine the performance impact of each layer on a set of benchmark applications.

5.6.1 Basic operation performance

Table 5.1 summarizes the basic operation latencies for each of the four layers for an NV-heap running on DRAM. The value for `new in` is the time to allocate and deallocate a small region of memory. The three "reference" rows give the time to assign to each type of reference and then set it to NULL (Base does not provide weak references), excluding transaction and opening overheads for Tx and C-Tx. The "nop tx" is the time to execute an empty transaction on the two transaction

Table 5.1: **Basic operation latency for NV-heaps running on DRAM** Each layer adds overhead to the basic operation latencies. Latencies for `new in` are listed as <0.1 because of inconsistencies due to caching effects.

Layer	Base (μs)	Safe (μs)	Tx (μs)	C-Tx (μs)
`new in`	<0.1	<0.1	<0.1	<0.1
V-to-NV reference	0.03	0.05	0.11	0.13
NV-to-NV reference	0.03	0.05	0.17	0.25
weak NV-to-NV reference	n/a	0.05	0.20	0.25
nop tx	n/a	n/a	0.05	0.05
Open for read	n/a	n/a	0.17	0.26
Open for write	n/a	n/a	1.68	1.99

systems. "Open for read" and "Open for write" give the times to open an object for access but do not include the transaction overhead.

The features that Safe, Tx, and C-Tx provide add significant overheads, but each layer makes it significantly easier to build reliable, persistent data structures. For example, safe reference operations take between 2.2× and 4× longer when executing inside a transaction (Tx versus Safe). This extra cost in latency is due to the operation descriptors and epoch barriers required to provide atomicity and durability. However, writing the code to build a reliable, persistent data structure is considerably more difficult without durable transactions.

5.6.2 The price of safety

To understand the overhead of programming language support for NV-heaps, we have implemented our benchmarks (see Table 4.2), in the Base, Safe, Tx, and C-Tx layers. Figure 5.5 shows the performance of each benchmark relative to its Base implementation, and the absolute performance is given in operations per second in the Base bar for each benchmark.

For a single thread, Base provides the highest performance, but Safe results in only a modest performance hit for most applications. One exception is BTree, which makes extensive use of NV-to-NV and weak NV-to-NV references, resulting in 62% lower performance for Safe. Overall, this suggests that, for many

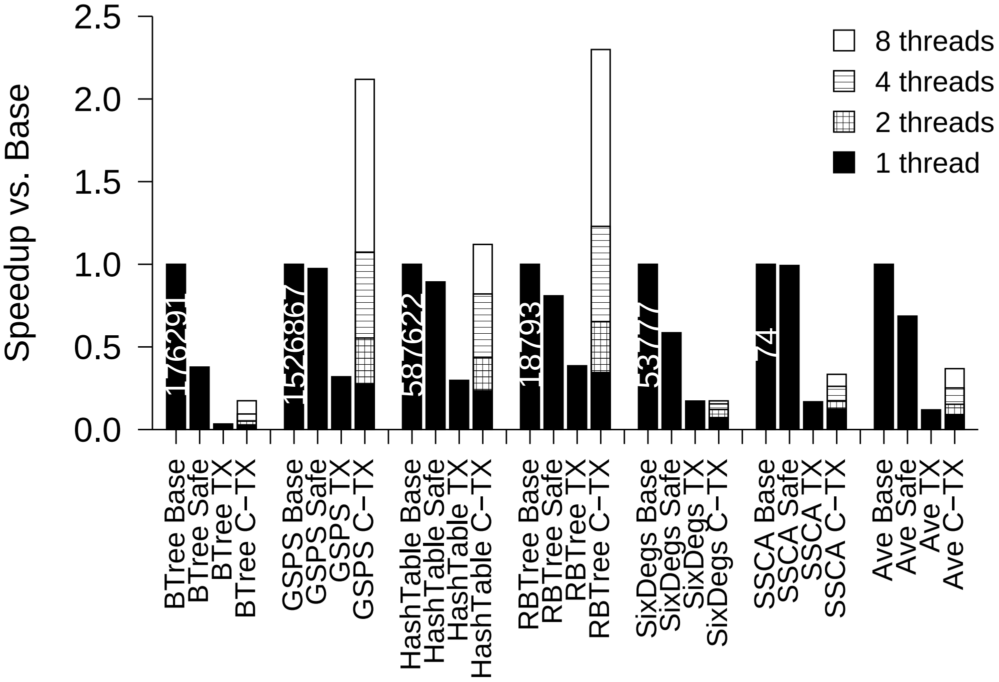


Figure 5.5: **The price of safety in NV-heaps** Removing safety guarantees improves performance by as much as 11×, but the resulting system is very difficult to use correctly. The different layers highlight the safety and performance trade-off.

applications, the extra safety that Safe provides will be worth the performance penalty.

Tx exacts a larger toll, reducing performance by 82% on average versus Safe. The cost of durability is due to the copying required to log objects in non-volatile memory. The price is especially steep for BTree, because writing a node of the tree into the log requires many reference copies.

Adding support for concurrency and conflict detection in C-Tx has a small effect on performance (30% on average relative to Tx), and allows HashTable, RBTree, and SPS to reclaim much of their lost performance.

The gap in performance between Base and C-Tx, 11× on average for single-threaded programs, is the cost of safety in our system. The increase in performance of Base is significant, but the price in terms of usability is high. The programmer must explicitly manage concurrency, atomicity, and failure recovery while avoiding the accidental creation of unsafe references. Whether this extra effort is worth the increased performance depends on the application and the amount of time the programmer is willing to spend writing, testing, and debugging the code. We found programming in this style to be tedious and error-prone.

5.7 Implementation limitations

This section describes some of the limitations in our current NV-heaps implementation. The language support proposed in this chapter has yet to be integrated into a compiler. Rather, NV-heaps is implemented as a library in C++ and requires the programmer to use the library interface correctly in order to provide strong safety guarantees. Unfortunately, a programmer can write code that circumvents the type safety provided by C++ and our library.

C/C++ do not prevent the programmer from potentially dangerous operations such as unsafe casting, manual pointer arithmetic, and exceeding the bounds of an array. These operations are especially pernicious in the NV-heaps environment because data corruption is permanent. For example, an NV-to-NV reference could be arbitrarily overwritten using a `void*` cast, creating an unsafe reference and potentially rendering the entire NV-heap useless.

The interface that NV-heaps provides makes a contract with the user about the safety of the data stored in the heap. The guarantees provided by each programming model require writing code that properly extends and uses the methods of the C++ classes we provide. For example, the programmer must use the NV-heaps base class and smart pointers, exposed by the BASE layer, to build persistent objects that can be properly allocated and freed from within the heap, accessed through V-to-NV and NV-to-NV references, and tracked by the NV-heap allocator. To guard against memory leaks, the SAFE layer also requires the use of our smart pointer types for all NV-heap references. Our smart pointers override the standard C/C++ reference, dereference, and casting operations. Using a regular C pointer to interact with an NV-heap breaks our type system and circumvents the the reference counting system. The transactional guarantees of TX and C-TX depend on building classes that inherit from a transactional base class (which inherits from the NV-heaps base class) and using our automatically-generated accessor methods for reading and modifying the fields of an object.

References in NV-heaps require special care. An NV-to-NV reference to an object whose type is of class `C` must be of type `C::NVPtr`, and it must never be declared outside of an object that inherits from the NV-heaps base class. Doing otherwise would create a non-volatile reference that lives in the volatile heap. If a failure occurs, such a reference would disappear but leave the object's reference count in an inconsistent state, making the object impossible to garbage collect. Any temporary references to an object of type `C` must be of type `C::VPtr` so that they prevent an object from being garbage collected while in use but evaporate in the case of a crash. However, `C::VPtr` type references must never be declared inside a persistent object derived from the NV-heaps base class because storing a V-to-NV reference in an NV-heap is unsafe.

It is critical to avoid creating temporary references that live on the stack because they cannot be garbage collected in the event of a failure. These temporary references affect an object's non-volatile reference count but are volatile. They can get created when passing pointers as arguments to functions or using pointers as the return types of functions. Functions should pass pointer arguments to an object of type `C` either by reference (`C::NVPtr &` or as V-to-NV pointers of type `C::VPtr`. Similarly, functions that need to return a pointer must only return pointers of type `C::VPtr`. Also, casting between `C::NVPtr` types is not allowed because it creates temporary references. Instead, a user must cast a `C::NVPtr` to a `C::VPtr` using a special `volatile_cast()` method.

In addition to getting references right, NV-heaps require the programmer to obey a set of rules when implementing classes that will be accessed transactionally (with the Tx or C-Tx programming models). A user-defined class must inherit from the transactional base class so that any instantiated objects can be correctly monitored and updated through the transactional memory system. The class must implement a factory method called `New()` to allocate an object in the NV-heap, and the class constructor must be private. Using a private constructor forces the programmer to explicitly create an object with the NV-heap allocator. Any fields defined in the class must be implemented with a set of macros that generate the transactional accessor methods (getters and setters). These methods implement the proper locking and logging operations to ensure ACID semantics.

The limitations of the current NV-heaps interface present a good opportunity for future work. As we will discuss in the next section, many of potential pitfalls can be avoided by pushing the programming language requirements into the compiler implementation.

5.8 Future work

The language support and programming models presented in this chapter provide strong safety guarantees for NV-heaps. However, we continue to find ways to improve on our existing implementation and provide more flexibility and robustness. In the remainder of this section, we discuss several important areas of future work including compiler support, safer garbage collection, and a new language feature to assist in providing fine-grained consistency guarantees.

5.8.1 Compiler support

To get the full benefit of language support for NV-heaps, our Java-like language and static dependent type system should be implemented in a compiler. Compiler support will codify the requirements of NV-heaps in the language. As we point out in Section 5.7, there are several limitations with our library-based C++ implementation. Compiler support would remove some of these potential safety violations and convert others from run-time exceptions to compile-time errors.

There are a range of options in implementing compiler support for NV-heaps. If we want to maintain the existing C++ interface, we could build a Lint-like [Joh78] program checker that would flag suspicious code that might violate the rules of the NV-heaps interface. This has the advantage of code re-use but there may be limitations regarding the types of illegal code we can catch without restricting the actual language syntax.

Another option is to build a source-to-source translator, using a compiler toolchain such as LLVM [LA04], to convert code written in our Java-like language to the existing C++ interface for NV-heaps. This also has the advantage of code re-use, and it has the additional benefit of controlling exactly the type of C++ source code that gets emitted.

Finally, we could augment an existing language such as Java to support NV-heaps-style semantics. This has the advantage of being compatible with an existing language, and it would integrate well with existing code bases. However, extending Java requires significant modifications to both the compiler and the run-time system. The current NV-heaps implementation would need to be rewritten to work within the Java Virtual Machine (JVM). It is unclear what sort of performance we could achieve running inside of the JVM.

5.8.2 Safe garbage collection

Garbage collection in NV-heaps is challenging because it must be robust against failures, it must be scalable, and it should perform very well. We chose a reference counting scheme because objects get collected in an incremental fashion as soon as they become dead. While this is a scalable and deterministic technique, it has a few important drawbacks. First, reference counts require frequent updates (i.e., whenever pointers are manipulated) and the updates must be thread-safe. In a persistent setting, this cost is greater because the updates must be durable, meaning the system waits until they reach non-volatile memory. Second, and perhaps more important, reference counting by itself cannot garbage collect cyclic data structures. Once a programmer creates a cycle, that data can never be collected. In NV-heaps, we solve this problem by providing weak references for cyclic pointers, but weak references work only if the programmer uses them correctly. So while weak references provide some guarantee that storage will be reclaimed, this guarantee is not as strong as it could be because it depends on the programmer to get it right.

One promising option to make garbage collection safer in NV-heaps is to augment our reference counting implementation to automatically collect cyclic garbage. Decades ago, a common approach was to use a backup mark and sweep collector, but the development, maintenance, and overhead costs were high. As a result, many researchers focused on specialized algorithms for cycle collectors [Lin92, MWL90, BR01, PBK+07] which improve running times and worst-case complexity. In the NV-heaps case, a cycle collector would need to detect and collect cyclic garbage in a transactional manner. Making the operations of such an algorithm recoverable may prove challenging given that they need to be implemented with very limited support (i.e., operation descriptors).

Alternatively, we could develop a tracing garbage collector for NV-heaps. To meet our scalability requirement, the scheme should avoid scanning the entire heap, which was a limitation of some previous persistent object stores [KW93, ONG93]. More recent work [CKWZ96, MMH96] partitioned the heap into independently collectible areas to limit tracing to small fractions of the heap, and [ML97] presented a scheme to reduce the overhead of inter-partition references and collect cycles that span partitions while preserving the localized nature of partitioned collection. These schemes, however, were designed with disk as the backing store for persistent objects. NV-heaps will require an implementation that optimizes for fast, non-volatile memories, which may look very different from previous approaches. The key challenge is minimizing and eliminating overheads while providing fully automatic collection.

5.8.3 Persistent keyword

To ensure that updates reach non-volatile storage and are permanent, NV-heaps provides an epoch barrier operation [CNF+09]. Epoch barriers order updates to storage, giving the programmer a way to reason about the state of data at a particular point in a program. A common idiom to guarantee consistency is to include a “valid” bit in a data structure, where a set valid bit indicates something special about the state of the rest of the data (like Dekker’s algorithm for mutual exclusion [Dij02]). For example, in NV-heaps, we use a valid bit per operation descriptor to indicate whether or not the operation has been written to the descriptor’s log. At recovery, if the valid bit is set, then the NV-heap will replay the operation descriptor, copying the logged contents to their in-place locations.

This programming idiom guarantees consistency as long as the programmer inserts the epoch barriers in the correct locations in the code. Consider the following code snippet to complete a memory allocation operation.

```
...
// Fill out the operation descriptor
opDescriptor->dstPtr = dstPtr;
opDescriptor->allocatedSpace = allocatedSpace;
EpochBarrier();
// Mark it as valid
opDescriptor->valid = true;
EpochBarrier();
// Play the operation descriptor
*(opDescriptor->dstPtr) = opDescriptor->allocatedSpace;
EpochBarrier();
// Mark it as invalid
opDescriptor->valid = false;
EpochBarrier();
...
```

It is essential that an epoch barrier appear after filling out the operation descriptor, after setting the valid bit, after playing the operation descriptor, and finally after clearing the valid bit. Leaving the epoch barrier out of any of these locations makes the operation potentially unsafe. For example, if the valid bit is marked true before the `dstPtr` field of the operation descriptor gets assigned (which could easily happen due to caching effects), then the system would assume that the field holds a valid pointer value even though it may be garbage.

To avoid these problems, we could extend our language to support a `persistent` keyword that describes the type of a variable as one that must be updated in a consistent and durable manner. This means that all non-volatile memory operations appearing before (in program order) an update to a `persistent` variable have completed and that any non-volatile memory operations appearing after the update to a `persistent` variable can occur only after the update has been written to memory. The compiler would automatically generate epoch barriers before and after updating the variable, guaranteeing consistency and durability. In the previous example, we would declare the valid field of the operation descriptor `persistent`. This eliminates many potential programmer errors that might result when using (or forgetting) epoch barriers.

5.9 Summary

In this chapter, we have described a set of execution models and programming language support for NV-heaps that allow programmers to build fast, safe, and scalable persistent data structures. We evaluated the trade-offs between safety, ease of use, and performance that each of our programming models provides. We also discussed limitations in our library-based implementation and proposed several topics for future work. In the next and final chapter, we will summarize the contributions of NV-heaps and the other work presented in this dissertation.

Acknowledgments

This chapter contains material from "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories", by Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which appears in *ASPLOS '11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems.* The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2011 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

This chapter contains material from "Programming Language Support for Fast, Byte-Addressable, Non-Volatile Memory", by Joel Coburn, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which will be submitted for publication. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Summary

The amount of data that we generate as a society is growing at an exponential rate. Our ability to store and analyze this data has become limited by the performance of conventional storage technologies. However, new non-volatile memory technologies promise to deliver DRAM-like performance to meet our data processing demands. These technologies will fundamentally change the way we design computer systems and interact with storage. In fact, this change is already evident in systems that use highly-optimized software stacks and specialized interfaces for flash-memory based SSDs [fus, vir, PRZ08, ONW$^+$11].

These technologies can integrate seamlessly into the storage hierarchy, be made directly accessible as memory, and can serve as a building block in distributed storage systems. We began this dissertation by characterizing these new technologies and their role in future storage devices. Using technologies such as PCM, we have shown that we can build storage devices accessible over PCIe interconnect or the processor's memory bus. These low latency, high bandwidth connections to storage help expose the full flexibility and performance of these new memory technologies. However, fully realizing these benefits requires us to rid software of disk-centric optimizations, design decisions, and architectures that limit performance and ignore bottlenecks that the poor performance of disks have hidden in the past.

This dissertation has focused on one of the most important aspects of storage: providing strong consistency guarantees. These guarantees allow us to reason about when and in what order writes to storage become durable. The software and algorithms that current applications, particularly databases, use to enforce strong consistency guarantees are critical to system performance, and most solutions are tightly coupled with disk technology. We have shown that by rethinking existing transaction mechanisms, we can engineer systems to provide both safety and high performance. We have also shown that, with fast storage, we can move from a block-based abstraction to a model that allows us to directly access storage in the same way we access volatile memory. This is an incredibly powerful abstraction because we can take advantage of the features of modern programming languages. However, it requires software support to provide safety both from system failures and from programmer errors that can permanently corrupt data.

Over the course of this dissertation, we have presented transactional support for fast, non-volatile memories that exploits the raw performance of these technologies while providing strong consistency guarantees. This includes a write-ahead logging scheme, hardware support for atomic write operations, a persistent object store for storage on the memory bus, and language support for persistent objects.

In Chapter 3, we introduced MARS, a new write-ahead logging scheme for fast, non-volatile memories that supports ACID transactions in an advanced SSD prototype. MARS is designed to be a replacement for ARIES, a popular algorithm for transactions used by many commercial databases. MARS utilizes a novel multi-part atomic write operation that takes advantage of the parallelism and performance of fast, non-volatile memories. Multi-part atomic writes can also be used to implement transactions for other applications, including persistent data structures such as key-value stores. Compared to executing transactions in software alone, our system increases effective bandwidth by up to 3.8× and decreases latency by 2.9×. MARS outperforms a baseline implementation of ARIES, which requires both redo and undo logging, by 3.7×.

In Chapter 4, we presented NV-heaps, a system designed for building fast and safe persistent data structures in storage attached to the processor's memory bus. NV-heaps provides an intuitive and familiar programming model. We have shown how NV-heaps protects against application and system failures by avoiding familiar programming bugs (i.e., dangling pointers) as well as new types of pointer safety bugs that arise only with persistent objects. We described the implementation of persistent objects, specialized pointers, automatic memory management, and atomic sections. For a variety of persistent data structures, NV-heaps outperforms BerkeleyDB and Stasis by 32× and 244× by avoiding the operating system and other software overheads. In Chapter 5, we presented programming language support for NV-heaps. We have shown that we can provide stronger safety guarantees by moving the implementation from a library into the compiler. We also presented several different programming models that can trade off safety, ease of use, and performance.

Efficient transaction mechanisms are critical to exploiting the full benefit of fast, non-volatile memories and to bringing them into the mainstream of computer storage. These versatile technologies may play various roles in the storage hierarchy depending on their cost, performance, density, etc. Given that, we presented system support for both a traditional, block-based interface to storage and a memory-like interface to storage. Both MARS and NV-heaps are attractive ways to provide fast and safe access to future storage architectures. They can serve as building blocks in redefining the role of persistence in our applications and meeting the growing demands of storage and data processing.

Acknowledgments

This chapter contains material from "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories", by Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which appears in *ASPLOS '11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems.* The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2011 by the Association for Computing Machinery, Inc. (ACM). Permission to make

This chapter contains material from "From ARIES to MARS: Reengineering Transaction Management for Next-Generation, Non-Volatile Memories", by Joel Coburn, Trevor Bunker, Rajesh K. Gupta, and Steven Swanson, which will be submitted for publication. The dissertation author was the primary investigator and author of this paper.

This chapter contains material from "Programming Language Support for Fast, Byte-Addressable, Non-Volatile Memory", by Joel Coburn, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson, which will be submitted for publication. The dissertation author was the primary investigator and author of this paper.

Bibliography

[AAK$^+$05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[ADJ$^+$96] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *SIGMOD Rec.*, 25(4):68–75, 1996.

[AH87] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 430–440, New York, NY, USA, 1987. ACM.

[Ake] Brian Aker. Libmemcached. http://libmemcached.org/.

[All09] Peter Allenbach. Java card 3: Classic functionality gets a connectivity boost, March 2009. http://java.sun.com/developer/technicalArticles/javacard/javacard3/.

[BBF$^+$08] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, pages 17–32, Washington, DC, USA, 2008. IEEE Computer Society.

[BDLM07] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 24–34, New York, NY, USA, 2007. ACM.

[bee] http://www.beecube.com/platform.html.

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[BM05] David A. Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *HiPC 2005: Proc. 12th International Conference on High Performance Computing*, pages 465–476, December 2005.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 117–128, New York, NY, USA, 2000. ACM.

[BO06] Rahul Biswas and Ed Ort. The java persistence api - a simpler programming model for entity persistence, May 2006. http://java.sun.com/developer/technicalArticles/J2EE/jpa/.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The gemstone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[BR01] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 207–235, London, UK, UK, 2001. Springer-Verlag.

[Bre08] Matthew J. Breitwisch. Phase change memory. *Interconnect Technology Conference, 2008. IITC 2008. International*, pages 219–221, June 2008.

[BRK+04] 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. *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 442–445, June 2004.

[BS77] Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

[Cat94] R. G. Cattell. *Object Data Management: Object-Oriented and Extended.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[CCA+11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven 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*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[CCM$^+$10] Adrian M. Caulfield, Joel Coburn, Todor I. Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[CDC$^+$10] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, pages 385–395, New York, NY, USA, 2010. ACM.

[CDG$^+$90] M. J. Carey, David J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The exodus extensible dbms project: an overview. pages 474–499, 1990.

[CEJ$^+$92] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9R1, HP Laboratories, November 1992.

[Chu] Steve Chu. Memcachedb. http://memcachedb.org/.

[CKWZ96] Jonathan E. Cook, Artur W. Klauser, Alexander L. Wolf, and Benjamin G. Zorn. Semi-automatic, self-adaptive control of garbage collection rates in object databases. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 377–388, New York, NY, USA, 1996. ACM.

[CL09] Sangyeun Cho and Hyunjin Lee. Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 347–357, New York, NY, USA, 2009. ACM.

[CME$^+$12] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven 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, ASPLOS '12, pages 387–400, New York, NY, USA, 2012. ACM.

[CNF+09] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960.

[Cor] Oracle Corporation. Zfs. http://hub.opensolaris.org/bin/view/Community+Group+zfs/.

[CSSB08] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 11–22, New York, NY, USA, 2008. ACM.

[DAR09] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: a hybrid pram and dram main memory system. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 664–469, New York, NY, USA, 2009. ACM.

[Dij02] Edsger W. Dijkstra. The origin of concurrent programming. chapter Cooperating sequential processes, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[dJKH93] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: a new approach to improving file systems. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 15–28, New York, NY, USA, 1993. ACM.

[DKM10] Alan Dearle, Graham N. C. Kirby, and Ron Morrison. Orthogonal persistence revisited. In *Proceedings of the Second international conference on Object databases*, ICOODB'09, pages 1–22, Berlin, Heidelberg, 2010. Springer-Verlag.

[DSPE08] B. Dieny, R. Sousa, G. Prenat, and U. Ebels. Spin-dependent phenomena and their implementation in spintronic devices. *VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on*, pages 70–71, April 2008.

[fus] http://www.fusionio.com/.

[GA01] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 70–80, New York, NY, USA, 2001. ACM.

[GC07] Justin E. Gottschlich and Daniel A. Connors. Dracostm: a practical c++ approach to software transactional memory. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 52–66, New York, NY, USA, 2007. ACM.

[GHKdJ96] R. Grimm, W.C. Hsieh, M.F. Kaashoek, and W. de Jonge. Atomic recovery units: failure atomicity for logical disks. *Distributed Computing Systems, International Conference on*, 0:26–37, 1996.

[HAG99] David M. Hansen, Daniel R. Adams, and Deborah K. Gracio. In the trenches with objectstore. *TAPOS*, 5(4):201–207, 1999.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[HLM94] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an nfs file server appliance. In *USENIX Winter*, pages 235–246, 1994.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[HMGJ04] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 73–84, New York, NY, USA, 2004. ACM.

[HMSC87] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in quicksilver. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 107–108, New York, NY, USA, 1987. ACM.

[HSATH06] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 74–83, New York, NY, USA, 2006. ACM.

[HWC$^+$04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[ICN$^+$10] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 3–14, New York, NY, USA, 2010. ACM.

[ITR09] International technology roadmap for semiconductors: Emerging research devices, 2009.

[jed09] Jedec standard: Low power double data rate 2 (lpddr2), March 2009.

[JLR$^+$94] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 48–59, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[Joh78] S. C. Johnson. Lint, a c program checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[JPS$^+$10] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, 3:681–692, September 2010.

[KTM$^+$08] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Young Min Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno. 2 mb spram (spin-transfer torque ram) with bit-by-bit bi-directional current write and parallelizing-direction current read. *Solid-State Circuits, IEEE Journal of*, 43(1):109–120, Jan. 2008.

[KW93] Elliot K. Kolodner and William E. Weihl. Atomic incremental garbage collection and recovery for a large stable heap. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 177–186, New York, NY, USA, 1993. ACM.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[LAC+96] 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 *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329, New York, NY, USA, 1996. ACM.

[LC97] David E. Lowell and Peter M. Chen. Free transactions with rio vista. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 92–101, New York, NY, USA, 1997. ACM.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[LIMB09] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[Lin92] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett.*, 44(4):215–220, December 1992.

[Lis88] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. *Commun. ACM*, 34(10):50–63, 1991.

[mem] Memcached. http://memcached.org/.

[Met] Joachim Metz. E-mail and appointment falsification analysis analysis of e-mail and appointment falsification on microsoft outlook/exchange. http://sourceforge.net/projects/libpff/files/.

[MHL$^+$92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[mic] Repair an .ost or .pst file in outlook. http://office.microsoft.com/en-us/outlook/ha010563001033.aspx.

[ML97] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 313–323, New York, NY, USA, 1997. ACM.

[MMH96] J. Eliot B. Moss, David S. Munro, and Richard L. Hudson. Pmos: A complete and coarse-grained incremental garbage collector for persistent object stores. In *POS*, pages 140–150, 1996.

[MSH$^+$06] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing.* Jun 2006. Held in conjunction with PLDI 2006. Expanded version available as TR 893, Department of Computer Science, University of Rochester, March 2006.

[MWL90] A. D. Martínez, R. Wachenchauzer, and R. D. Lins. Cyclic reference counting with local mark-scan. *Inf. Process. Lett.*, 34(1):31–35, February 1990.

[MZB00] Alonso Marquez, John N. Zigman, and Stephen M. Blackburn. Fast portable orthogonally persistent java. *Softw. Pract. Exper.*, 30(4):449–479, 2000.

[ONG93] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 161–174, New York, NY, USA, 1993. ACM.

[ONW$^+$11] Xiangyong Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D.K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 301–311, February 2011.

[ora] Berkeley db. http://www.oracle.com/technology/products/berkeley-db/index.html.

[OTMW04] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *Proceedings of IFIP 3rd International Conference on Theoretical Computer Science*, pages 437–450, 2004.

[PBK+07] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. *ACM Trans. Program. Lang. Syst.*, 29(4), August 2007.

[PRZ08] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 147–160, Berkeley, CA, USA, 2008. USENIX Association.

[QKF+09a] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[QKF+09b] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[QSLF11] Moinuddin K. Qureshi, Andre Seznec, Luis A. Lastras, and Michele M. Franceschini. Practical and secure pcm systems by online detection of malicious write streams. *High-Performance Computer Architecture, International Symposium on*, 0:478–489, 2011.

[QSR09] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[RHL05] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[SA02] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the eros single-level store. In *ATEC '02: Proceedings of the General Track* *of the annual conference on USENIX Annual Technical Conference*, pages 59–72, Berkeley, CA, USA, 2002. USENIX Association.

[SATH$^+$06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[SB06] Russell Sears and Eric Brewer. Stasis: flexible transactional storage. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 29–44, Berkeley, CA, USA, 2006. USENIX Association.

[SB09] Russell Sears and Eric Brewer. Segment-based recovery: write-ahead logging revisited. *Proc. VLDB Endow.*, 2:490–501, August 2009.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In *POS*, pages 11–33, 1992.

[SLSB10] Stuart Schechter, Gabriel H. Loh, Karin Straus, and Doug Burger. Use ecp, not ecc, for hard failures in resistive memories. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 141–152, New York, NY, USA, 2010. ACM.

[SMK$^+$93] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 146–160, New York, NY, USA, 1993. ACM.

[SO92] Margo Seltzer and Michael Olson. Libtp: Portable, modular transactions for unix. In *Proceedings of the 1992 Winter Usenix*, pages 9–25, 1992.

[Sol96] Frank G. Soltis. *Inside the as/400.* Twenty Ninth Street Press, 1996.

[SPC01] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[SS05] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*

'05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, pages 240–248, New York, NY, USA, 2005. ACM.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[TKM+07] R. Takemura, T. Kawahara, K. Miura, J. Hayakawa, S. Ikeda, Y.M. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno. 2mb spram design: Bi-directional current write and parallelizing-direction current read schemes based on spin-transfer torque switching. *Integrated Circuit Design and Technology, 2007. ICICDT '07. IEEE International Conference on*, pages 1–4, 30 2007-June 1 2007.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, February 1997.

[vir] http://www.virident.com/.

[VTS11] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 2011. ACM.

[WD94] Seth J. White and David J. DeWitt. Quickstore: a high performance mapped object store. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 395–406, New York, NY, USA, 1994. ACM.

[xdd] Xdd version 6.5. http://www.ioperformance.com/.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.

[YMC+11] Doe Hyun Yoon, N. Muralimanohar, Jichuan Chang, P. Ranganathan, N.P. Jouppi, and M. Erez. Free-p: Protecting non-volatile memory against both hard and soft errors. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 466–477, feb. 2011.

[ZZYZ09] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 14–23, New York, NY, USA, 2009. ACM.