UNIVERSITY OF CALIFORNIA, SAN DIEGO

Moneta: A Storage System for Fast Non-Volatile Memories

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

in

Computer Science (Computer Engineering)

by

Adrian Michael Caulfield

Committee in charge:

Professor Steven Swanson, Chair
Professor Rajesh Gupta
Professor Paul Siegel
Professor Dean Tullsen
Professor Geoff Voelker

2013

The Dissertation of Adrian Michael Caulfield is approved and is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____
Chair

University of California, San Diego

2013

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Steven Swanson for his support as the chair of my committee. Throughout my entire graduate school experience, including many papers and long nights, his guidance and encouragement has brought the best out of me and helped me to achieve my goals.

I would also like to acknowledge the other members of the Non-volatile Systems Lab, without whom my research would have no doubt taken fives times as long. Special thanks to Todor Mollov, Joel Coburn, Ameen Akel, Arup De, Alex Eisner, and Trevor Bunker for their extensive help with the Moneta hardware.

Thanks also to Professor Goeff Voelker for his guidance and support on several of the key works that make up this dissertation. The rest of my committee also provided valueable comments and feedback, for which I am greatful.

I want to thank my parents and brother for their support and advice. They always made sure I had everything I needed throughout all of my schooling.

My wonderful wife, Laura Caulfield, has provided immesurable support and encouragement during our shared adventure through graduate school.

Chapters 1 and 2 and Appendix A contain material from “Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing”, by Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson, which appears in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC ’10). The dissertation author was the primary investigator and author of this paper. The material in Chapters 1 and 2 and Appendix A is copyright ©2010 by the IEEE.

Chapters 1, 2, and 5 and Appendix A contain material from “QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks”, by Adrian M. Caulfield and Steven Swanson, which appears in *ISCA '13: Proceeding of the 40th Annual International Symposium on Computer Architecture*. The dissertation author was the primary investigator and author of this paper. The material in Chapters 1, 2, and 5, and Appendix A is copyright ©2013 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, and 4 and Appendix A 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 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. The dissertation author was the primary investigator and author of this paper. The material in Chapters 1, 2, and 4 and Appendix A 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, and 3, and Appendix A 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, which appears in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (MICRO ’43). The dissertation author was the primary investigator and author of this paper. The material in Chapters 1, 2, and 3 and Appendix 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.

VITA

2007	Bachelor of Arts, University of Washington
2007–2013	Research Assistant, University of California, San Diego
2010	Master of Science, University of California, San Diego
2011	Candidate of Philosophy, University of California, San Diego
2013	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Adrian M. Caulfield and Steven Swanson. “QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks”, In *ISCA ’13: Proceeding of the 40th Annual International Symposium on Computer Architecture*, June 2013.

Adrian M. Caulfield, Todor I. Mollov, Louis Eisner, Arup De, Joel Coburn, and Steven Swanson. “Providing Safe, User Space Access to Fast, Solid State Disks”, In *Proceeding of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2012.

Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. “Onyx: A Prototype Phase-Change Memory Storage Array”, In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage’11, June 2011.

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 *ASPLOS ’11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2011.

Adrian M. Caulfield, Joel Coburn, Todor 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, 2010.

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 2010 43rd Annual IEEE/ACM*

International Symposium on Microarchitecture, MICRO ’43, 2010.

Sungjin Lee, Kermin Fleming, Jihoon Park, Keonsoo Ha, Adrian M. Caulfield, Steven Swanson, Arvind, and Jihong Kim. “BlueSSD: An Open Platform for Cross-layer Experiments for NAND Flash-based SSDs”, In *Proceedings of the 2010 Workshop on Architectural Research Prototyping*, 2010.

Laura M. Grupp and Adrian M. Caulfield and Joel Coburn and John Davis and Steven Swanson. “Beyond the Datasheet: Using Test Beds to Probe Non-Volatile Memories’ Dark Secrets”, In *IEEE Globecom 2010 Workshop on Application of Communication Theory to Emerging Memory Technologies (ACTEMT 2010)*, 2010.

Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. “Gordon: An Improved Architecture for Data-Intensive Applications”, *IEEE Micro*, 30(1):121–130, 2010.

Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. “Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications”, In *ASPLOS ’09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

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

ABSTRACT OF THE DISSERTATION

Moneta: A Storage System for Fast Non-Volatile Memories

by

Adrian Michael Caulfield

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2013

Professor Steven Swanson, Chair

Over the last few decades, storage performance has stagnated in comparison to the performance of the rest of the system. Over this time, system designers have continued to add additional layers of abstraction and optimization to the storage hierarchy in an attempt to hide and optimize accesses to high-latency storage. Emerging non-volatile memory technologies promise many orders of magnitude increases in storage array performance compared to existing storage technologies, but the thick layers of software built on the assumption that storage is slow risk squandering the full potential of these new devices.

This dissertation describes a prototype high-performance storage array, called Moneta, designed for next-generation non-volatile memories, such as phase-change memory, that offer near-DRAM performance. Moneta allows us to explore the architecture of the storage array, the impact of software overheads on performance, the effects of non-volatile technology parameters on bandwidth and latency, and the ultimate benefit to applications.

Using Moneta, we show that system software can be optimized to expose fast storage with minimal latency overheads. Moneta reduces software overheads by 62% for 4 KB operations through the operating system, speeding up a range of file system, paging, and database workloads by up to 8.7× compared to flash-based SSDs.

Moneta-Direct extends Moneta by refactoring trusted code throughout the IO stack. This allows applications to bypass the operating system and file system entirely on most accesses, further reducing IO overheads by 58% and increasing throughput by 7.6×. Moneta-Direct demonstrates the importance of redesigning the IO stack to work efficiently with emerging fast non-volatile storage.

Finally, a further modification to Moneta allows us to explore the performance of distributed storage networks. These networks are integral parts of building scalable storage solutions. By integrating a low-latency network directly into the SSD, we can reduce the costs of accessing remote storage by up to 95% compared to commonly used higher-latency remote storage and network protocol layers.

Overall, Moneta demonstrates the critical need to continue to redesign system architectures to make the best use of fast non-volatile memory technologies.

Chapter 1

Introduction

Emerging fast, non-volatile technologies such as phase change, spin-transfer torque, and memristor memories make it possible to build storage devices that are orders of magnitude faster than even the fastest flash-based solid-state disks (SSDs). These technologies will rewrite the rules governing how storage hardware and software interact to determine overall storage system performance. In particular, software overheads that used to contribute marginally to latency (because storage hardware was slow) will potentially squander the performance that these new memories can provide.

For many years, the performance of persistent storage (i.e., disks) has lagged far behind that of microprocessors. Since 1970, microprocessor performance grew by roughly 200,000×. During the same period, disk access latency has fallen by only 9× while bandwidth has risen by only 163× [104, 43].

The emergence of non-volatile, solid-state memories (such as NAND flash and phase-change memories, among others) has signaled the beginning of the end for painfully slow non-volatile storage. These technologies will potentially reduce latency and increase bandwidth for non-volatile storage by many orders of magnitude, but fully harnessing their performance will require overcoming the legacy of disk-based storage systems. Chapter 2 provides some background on the various storage technologies and the trends driven by these new technologies.

Emerging storage technologies beyond flash have several note-worthy benefits when compared against both disk drives and flash based SSDs. First, the performance potential of the new technologies is far greater than both disks and flash, with SSD level accesses shrinking to just a few microseconds, compared to hundreds of microseconds for flash and milliseconds for disk based storage. Second, these devices have much higher endurance than flash memory and require very limited management and wear-leveling algorithms. Finally, the new memories offer the possibility of byte-addressable, in-place updates, potentially changing the way we access and treat storage. In many ways, one can think of these new technologies as non-volatile DRAM, with similar interfaces and projected performance within 2-3 × of DRAM.

In order to fully exploit these technologies, we must overcome the decades of hardware and software design decisions that assume that storage is slow. The hardware interfaces that connect individual disks to computer systems are sluggish (∼300 MB/s for SATA II and SAS, 600 MB/s for SATA 600) and connect to the slower "south bridge" portion of the CPU chip set [46]. RAID controllers connect via high-bandwidth PCIe, but the low-performance, general-purpose microprocessors they use to schedule IO requests limit their throughput and add latency [38].

Software also limits IO performance. Overheads in the operating system's IO stack are large enough that, for solid-state storage technologies, they can exceed the hardware access time. Since it takes ∼20,000 instructions to issue and complete a 4 KB IO request under standard Linux, the computational overhead of performing hundreds of thousands of IO requests per second can limit both IO and application performance.

This dissertation explores the design of both a prototype storage array, called Moneta, targeting these emerging non-volatile memory technologies, as well as the system-software design necessary to fully utilize such a device. Moneta allows us to explore the architecture of the storage array, the impact of software overheads on performance, the effects of non-volatile technology parameters on bandwidth and latency, and the ultimate benefit to applications. We have implemented Moneta using a PCIe-attached array of FPGAs and DRAM. The FPGAs implement a scheduler and a distributed set of configurable memory controllers. The controllers allow us to emulate fast non-volatile memories by accurately modeling memory technology parameters such as device read and write times, array geometry, and internal buffering. The Moneta hardware design is discussed through the dissertation and covered in extensive detail in Appendix A.

Achieving high performance in Moneta requires simultaneously optimizing its hardware and software components. In Chapter 3, we characterize the overheads in the existing Linux IO stack in detail, and show that a redesigned IO stack combined with an optimized hardware/software interface reduces IO latency by nearly 2× and increases bandwidth by up to 18×. Tuning the Moneta hardware improves bandwidth by an additional 75% for some workloads.

Chapter 3 also explores the impact of non-volatile memory performance and organization on Moneta's performance and energy efficiency. Using a range of IO benchmarks and applications, we characterize Moneta's performance and energy efficiency, and compare them to several other storage arrays.

While Chapter 3 explores the operating system level performance of Moneta, application and file-system performance still leave much room for improvement. Chapter 4 introduces Moneta Direct (Moneta-D), a second iteration of our prototype SSD that removes operating- and file-system costs by transparently bypassing both, while preserving their management and protection functions. These costs account for as much as 30% of the total latency of a 4 KB request, and can reduce sustained throughput by 85%.

Using a virtualized device interface, untrusted user space library and refactored kernel I/O stack, Moneta-D allows any unmodified application to directly talk to the storage array for most file data accesses. Moneta-D maintains the operating and file-system control over protection policy by moving the policy enforcement into the hardware, but requiring the trusted operating system code to setup and maintain the protection policy for the hardware.

Chapter 4 also includes an evaluation of the performance of Moneta-D and some of the key design decisions it includes. For example, we explore several ways in which Moneta-D trades off between CPU overhead and performance using different notification techniques to signal to applications that I/O requests have completed. We use a set of I/O benchmarks and database workloads and show that Moneta-D provides up to 5.7× performance improvements for small databases and can reduce small I/O overheads by as much as 64%.

Modern storage systems rely on complex software and interconnects to provide scalable, reliable access to large amounts of data across multiple machines. In conventional, disk-based storage systems the overheads from file systems, remote block device protocols (e.g., iSCSI and Fibre Channel), and network stacks are tiny compared to the storage media access time, so software overheads do not limit scalability or performance.

Systems like Moneta and Moneta-D change this landscape completely by dramatically improving storage media performance. As a result, software shifts from being the least important component in overall storage system performance to being a critical bottleneck.

The software costs in scalable storage systems arise because, until recently, designers could freely compose existing system components to implement new and useful storage system features. For instance, the software stack running on a typical, commodity storage area network (SAN) will include a file system (e.g., GFS [94]), a logical volume manager (e.g., LVM), remote block transport layer client (e.g., the iSCSI initiator), the client side TCP/IP network stack, the server network stack, the remote block device server (e.g., the iSCSI target), and the block device driver. The combination of these layers (in addition to the other operating system overheads) adds 288 μs of latency to a 4 kB read. Hardware accelerated solutions (e.g., Fibre Channel) eliminate some of these costs, but still add over 90 μs per access.

Chapter 5 extends Moneta's low latency direct, user-space access into large distributed storage area networks. It describes a new SAN architecture called *QuickSAN*. QuickSAN re-examines the hardware and software structure of distributed storage systems to minimize software overheads and let the performance of the underlying storage technology shine through.

QuickSAN improves SAN performance in two ways. First, it provides a very low-latency, hardware-accelerated block transport mechanism (based on 10 Gbit ethernet) and integrates it directly into an SSD. Second, it extends Moneta-D's OS bypass mechanism to allow an application to access remote data without (in most cases) any intervention from the operating system while still enforcing file system protections.

Finally, Chapter 6 concludes this dissertation and summarizes the findings from the preceding chapters.

Acknowledgements

This chapter contains material from “Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing”, by Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson, which appears in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC ’10). The dissertation author was the primary investigator and author of this paper.

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, which appears in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (MICRO '43). The dissertation author was the primary 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 "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 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. The dissertation author was the primary 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 “QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks”, by Adrian M. Caulfield and Steven Swanson, which appears in *ISCA ’13: Proceeding of the 40th Annual International Symposium on Computer Architecture*. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2013 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 2

Technologies and Trends

This chapter provides background on current and future storage technologies, and the trends in performance that motivate the rest of the work in this dissertation. We begin with brief introductions to the relevant technologies, and then discuss the broader bandwidth, latency, and energy trends present across the devices.

2.1 Hard Disks

Hard disks have been the mass storage technology of choice for the last four decades. Disks provide high-density, high-reliability storage with good sequential access performance. However, disk performance has not kept pace with increases in processor performance. Since 1970 processor performance has increased nearly 200,000× while disk bandwidth increased by 163× and disk latency improved by only 9× [43, 104].

Disks store data on stacks of rotating magnetic platters. Disks store and retrieve data from each side of the platters using a sensor, or head, on the end of an arm. The head detects the polarity of the magnetic field on the platter and translates it into bits of data. Similarly, during a write, the head alters the polarity of the field to store data. The arm moves linearly across the platters and, in combination with the rotation of the platters, allows data storage at all points on the surface. The delay during the linear movement of the arm is known as the seek latency of a disk drive, while the disk rotation is known as the rotational delay.

The notable characteristics of disks for this dissertation are their large access latencies (typically 5 to 7 ms) resulting from rotational delay and linear arm movement (seek time) as the head pans to the correct track. Disk bandwidth is also an important characteristic: bandwidth for a single disk ranges between 500 KB/s on random accesses to 138 MB/s during large sequential accesses [106].

Disk characteristics are now well understood and optimizations throughout the IO stack make performance slightly better, but most applications treat storage as something to access as infrequently as possible. For those applications that must access large amounts of data, such as databases, disk performance severely limits application performance. More recent memory technologies eliminate the fundamentally slow aspects of disk storage since they have no moving parts enabling uniform access latencies regardless of the ordering of requests.

2.2 NAND Flash Memory

Flash memory, specifically NAND flash memory, is a non-volatile solid state storage technology. Flash has been around for many years, but only within the last 5 to 7 years has its density increased sufficiently to make it a viable primary storage medium. NAND flash forms the basis of most commercial solid-state storage devices including solid-state disks, USB “thumb” drives, and storage media for cameras and portable devices.

Flash stores bits by trapping electrons on a ”floating” gate in a transistor. The trapped charge alters the voltage required to ”turn on” the transistor. To trap electrons on the gate, a large voltage is applied, causing electrons to tunnel through the insulator onto the gate. Current flash can use two, four, or eight charge levels on the floating gate. In the first case, each cell stores one bit of data, this is known as a Single Level Cell (SLC).

The later two are Multi-Level Cell (MLC) and Triple-Level Cell (TLC) and store two and three bits per cell, respectively.

NAND flash typically groups from 2 KB to 8 KB into a page and then 64 to 256 pages into a block. Reads and programs operate at page granularity, while erases occur at the block level. To maintain data integrity flash requires that each page receives at most one programming operation between erases.

Programming and erasing flash puts significant stress on the cells, which can eventually cause programming or erasing to fail. Modern NAND flash chips have a maximum erase count of between 3,000 (TLC) and 100,000 (SLC) cycles per block. This is the number of erase-program cycles that the chip can sustain before an unacceptable level of wear occurs and too many bit errors occur.

Flash Translation Layers (FTL) provide a wear management scheme to balance the number of erases each block performs. The FTL adds a layer of indirection between the sectors requested by the operating system and the physical address of the data stored in memory. This enables blocks of data to be moved around on the flash device to distribute wear. Without the FTL, blocks could potentially wear out at uneven rates, necessitating capacity decreases or device failure.

Several types of mass storage devices using flash memory are currently available. One is a direct hard disk replacement, connecting to a Serial ATA (SATA) or Serially Attached SCSI (SAS) bus. The other connects flash memory to a PCI express (PCIe) bus, enabling higher bandwidth and lower latency accesses [29, 47].

We have measured PCIe attached flash SSDs sustaining latencies of 68 μs and bandwidth of 250 MB/s on a random 4 KB read workload.

In summary, the main differences between flash memory and disks are three fold. First, the solid state nature of flash eliminates the address dependent seek latencies found in disks due to the mechanical motion of the disk arm. Second, flash requires the use of a FTL to manage the lifetime of each of the blocks of data in a flash device. Finally, access latencies in flash are operation dependent, affecting the way requests should be scheduled for flash memory.

2.3 Phase Change RAM

Phase Change Random Access Memory (PCRAM) stores bits not with electrical charge, but by manipulating the resistive properties of a material. To program a bit, PCRAM memory heats a small amount of chalcogenide (the material used in recordable CD-ROMs) and then cools it either slowly or quickly, depending on the value being programmed [9]. The different cooling rates create different structural patterns in the material, each with differing electrical resistance. To read a bit back, PCRAM passes current through the material, and depending on how quickly the chalcogenide cooled, encounters either high resistance or low resistance. PCRAM is bit alterable, meaning that each of the bits can be toggled independently, removing the need for the extra erase operation needed for flash memory.

PCRAM is projected to eventually match DRAM's latency and bandwidth, continuing recent trends of fast density and performance increases, and may become a viable technology to replace DRAM as a main memory technology [53]. The ITRS roadmap projects a PCRAM write time of less than 50 ns by 2024 [2]. For this work we assume PCRAM latencies of 69.5 ns and 215 ns as projected by [6].

PCRAM's bit alterability makes it better suited for replacing main memory in systems. However, since writes to storage often occur in larger blocks, having byte addressable memory is not necessarily a benefit when used as storage, especially if it incurs additional costs.

PCRAM removes many of the restrictions that flash imposes and eases others. Unlike Flash, PCRAM is byte addressable and requires no erase operation. Although PCRAM's durability is 10-100×that of flash, it still requires some wear levelling. PCRAM's lack of an erase operation and hence the avoidance of the read/write vs erase granularity disparity found in flash, makes wear levelling in PCRAM much simpler. Schemes such as start-gap [77] transparently handle wear levelling with minimal overhead. Similar to flash, PCRAM's write operations are longer than its reads, however the differences are much smaller (less than 4×). In most respects PCRAM is best thought of as a slightly slower DRAM.

2.4 Spin-Torque MRAM and the Memristor

Other technologies such as Spin-Torque MRAMs and the Memristor are also on the horizon, albeit a few years farther out than PCRAM. These technologies promise most of the same benefits as PCRAM with lower energy costs, higher endurance and improved latencies and bandwidth.

ST-MRAM uses layers of magnetic material that can be easily integrated with traditional chip production processes. Stacking a fixed layer of magnetic material with a free magnetic layer creates a magnetic tunnel junction (MJT). By changing the orientation of the free layer the resistance of the MTJ can be altered depending on whether the magnetic moment of the layers are in a parallel (low resistance) or anti-parallel (high resistance) configuration. The first, low density sample parts will be available within a year, offering latencies of 35 ns for both reads and writes [27].

The Memristor is another promising future non-volatile storage technology. Memristors are a passive circuit element that alters its resistance based on the direction of current flow through the device. When current ceases, the resistance is remembered. By building arrays of these circuit elements we can construct storage devices just like ST-MRAMs and PCRAM [40]. These devices are much less mature than either PCRAM or ST-MRAM so their characteristics are less clear, but they should be useful in many of

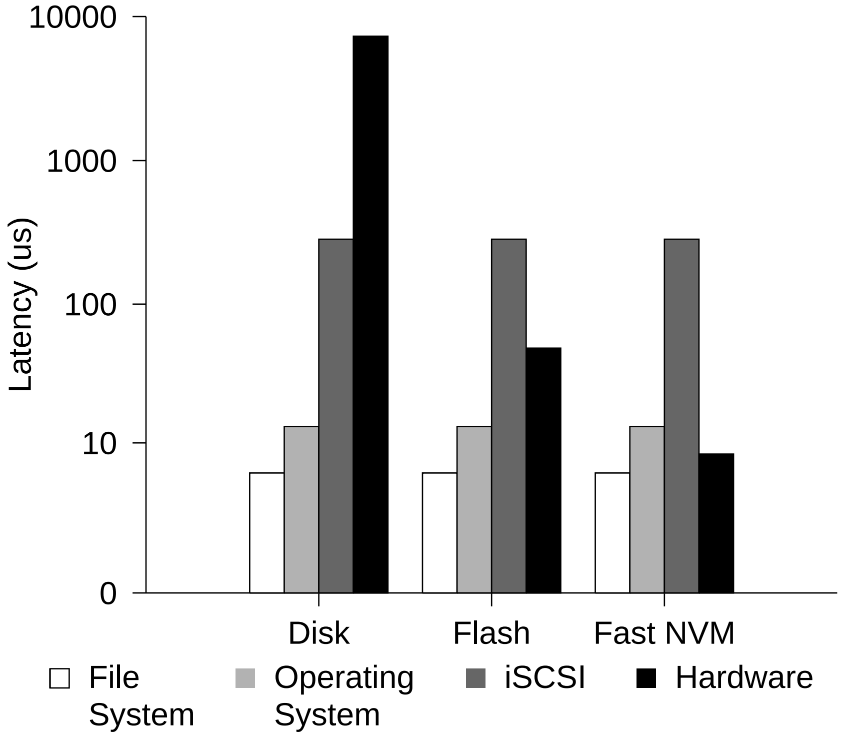

Figure 2.1. As the hardware access latency for storage technologies shrinks, software overheads will become the dominant latency cost in accessing storage.

the same applications.

2.5 Trends in Storage Technology

The new memory technologies introduced in the preceding sections show multiple paths forward towards non-volatile memory technologies that come within a small factor of DRAM performance. These technologies will drive storage latency from the 7 ms average hard disk access time to 8-10 μs with a device like the one described in this dissertation based on PCRAM or ST-MRAM.

Figure 2.1 illustrates the effect of decreasing hardware latencies. It shows several of the main components of I/O device latency: the file system, operating system, iSCSI latency (for remote storage accesses), and the hardware latency. Hardware latencies

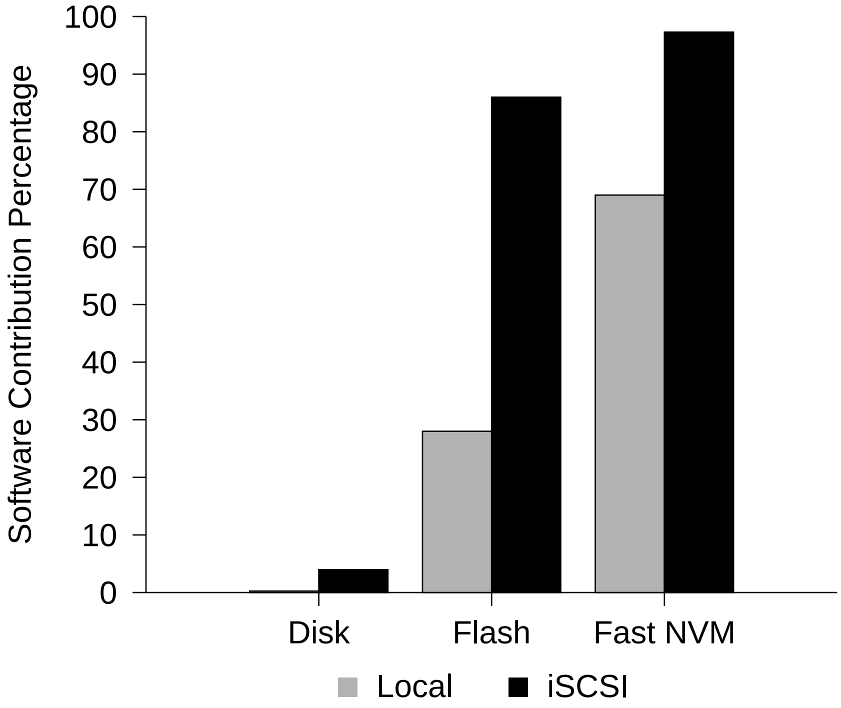

Figure 2.2. The graph shows the percentage of the total request latency contributed by software, for both local accesses and remote accesses through a software iSCSI stack. As memory latency decreases, software's contribution becomes larger, accounting for as much as 97% of the total latency.

for flash based storage are about two orders of magnitude smaller than for disks, but software overheads remain the same. This means as storage gets faster, the percentage of an I/O request accounted for by software components is increasing rapidly - from about 4% for disks to 84% for flash. A further reduction in latency from technologies such as PCRAM and ST-MRAM will drive software latencies to account for as much as 97% of the total request latency. Figure 2.2 depicts this trend, showing the software overheads as a percentage of the total request latency.

Acknowledgements

This chapter contains material from “Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing”, by Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson, which appears in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC ’10). The dissertation author was the primary investigator and author of this paper.

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, which appears in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (MICRO ’43). The dissertation author was the primary 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 “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*

17th International Conference on Architectural Support for Programming Languages and Operating Systems. The dissertation author was the primary 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 “QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks”, by Adrian M. Caulfield and Steven Swanson, which appears in *ISCA ’13: Proceeding of the 40th Annual International Symposium on Computer Architecture*. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2013 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 3

The Moneta Kernel Stack

The Moneta prototype SSD introduced in the introduction offers the potential to dramatically alter the storage performance seen by the operating system. By replacing spinning disks and flash based storage, with latencies of 7 ms and 100s of microseconds respectively, with emerging fast non-volatile memories that are orders of magnitude faster, Moneta offers much potential, but decades of legacy storage system designs pose a risk of squandering any significant gains.

Over the last four decades, storage system designs were driven by the fundamental assumption that storage is slow. This led to the addition of many layers of latency intensive software optimizations, to reorder, schedule, and prioritize the sequence of IO requests sent the hardware. Even though these optimizations potentially slow individual accesses by 100s of microseconds, they still account for less than a few percent of the total IO request latency, and often reduce the total latency of a sequence of requests.

This chapter covers the kernel level design and implementation of the Moneta system. We first characterize the overheads in the existing Linux IO stack in detail, and show that a redesigned IO stack combined with an optimized hardware/software interface reduces IO latency by nearly 2× and increases bandwidth by up to 18×. Tuning the Moneta hardware improves bandwidth by an additional 75% for some workloads.

We present two findings on the impact of non-volatile memory performance and organization on Moneta's performance and energy efficiency. First, some optimizations that improve PCM's performance and energy efficiency as a main memory technology do not apply in storage applications because of different usage patterns and requirements. Second, for 4 KB accesses, Moneta provides enough internal parallelism to completely hide memory access times of up to 1 μs, suggesting that memory designers could safely trade off performance for density in memory devices targeted at storage applications.

Results for a range of IO benchmarks demonstrate that Moneta outperforms existing storage technologies by a wide margin. Moneta can sustain up to 2.2 GB/s on random 4 KB accesses, compared to 250 MB/s for a state-of-the-art flash-based SSD. It can also sustain over 1.1 M 512-byte random IO operations per second. While Moneta is nearly 10× faster than the flash drive, software overhead beyond the IO stack (e.g., in the file system and in application) limit application-level speedups: Compared to the same flash drive, Moneta speeds up applications by a harmonic mean of just 2.1×, demonstrating that further work is necessary to fully realize Moneta's potential at the application level.

The remainder of the chapter is organized as follows. Section 3.1 briefly describes the Moneta prototype. Sections 3.2, 3.3, and 3.4 analyze the baseline Moneta system performance and describe software and hardware optimizations. Section 3.5 compares Moneta to existing storage technologies, and Section 3.6 describes related work. Finally, Section 3.7 summarizes this chapter.

3.1 The Moneta prototype

This section describes the baseline Moneta architecture and the hardware system we use to emulate it. Appendix A contains more implementation details of the Moneta prototype.

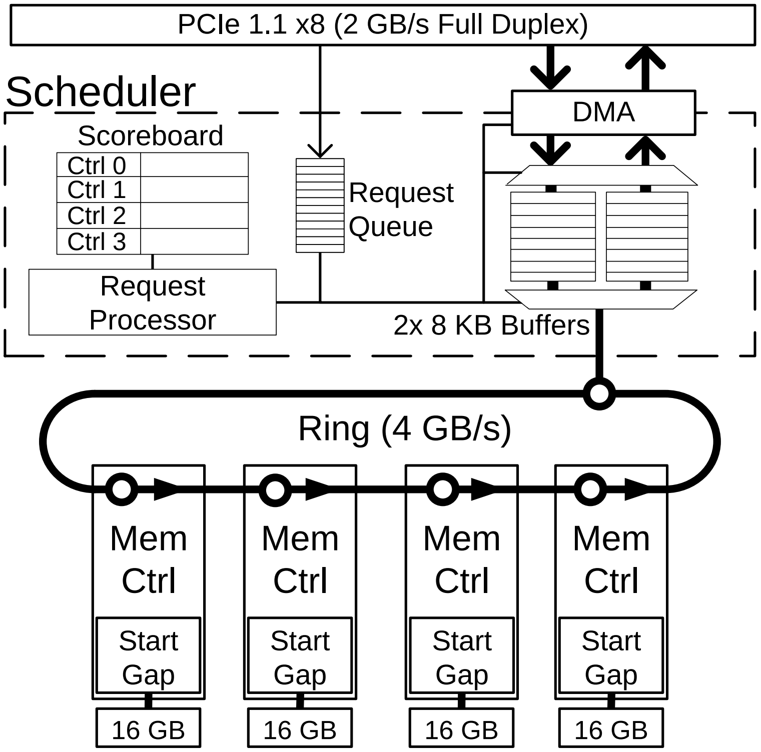

Figure 3.1. Moneta's PCM memory controllers connect to the scheduler via a 4 GB/s ring. A 2 GB/s full duplex PCIe link connects the scheduler and DMA engine to the host. The scheduler manages 8 KB buffers as it processes IO requests in FIFO order. A scoreboard tracks the state of the memory controllers.

3.1.1 Moneta array architecture

Figure 3.1 shows the architecture of the Moneta storage array. Moneta's architecture provides low-latency access to a large amount of non-volatile memory spread across four memory controllers. A scheduler manages the entire array by coordinating data transfers over the PCIe interface and the ring-based network that connects the independent memory controllers to a set of input/output queues. Moneta attaches to the computer system via an eight-lane PCIe 1.1 interface that provides a 2 GB/s full-duplex connection (4 GB/s total).

The Moneta scheduler The scheduler orchestrates Moneta's operation. It contains a DMA controller and a Programmed IO (PIO) interface to communicate with the host machine, a set of internal buffers for incoming and outgoing data, several state machines, and an interface to the 128-bit token-ring network. The Moneta scheduler stripes internal storage addresses across the memory controllers to extract parallelism from large requests. The baseline stripe size is 8 KB.

Requests arrive on the PCIe interface as PIO writes from the software driver. Each request comprises three 64-bit words that contain a sector address, a DMA memory address (in host memory), a 32-bit transfer length, and a collection of control bits that includes a tag number and an op code (read or write). Sectors are 512 bytes. The tag is unique across outstanding requests and allows for multiple in-flight requests, similar to SATA's Native Command Queuing [42].

The scheduler places requests into a FIFO queue as they arrive and processes them in order. Depending on the request's size and alignment, the scheduler breaks it into one or more transfers of up to 8 KB. It then allocates a buffer for each transfer from two 8 KB buffers in the scheduler. If buffer space is not available, the scheduler stalls until another transfer completes.

For write transfers, the scheduler issues a DMA command to transfer data from the host's memory into its buffer. When the DMA transfer completes, the scheduler checks an internal scoreboard to determine whether the target memory controller has space to receive the data, and waits until sufficient space is available. Once the transfer completes, the scheduler's buffer is available for another transfer. The steps for a read transfer are similar except that the steps are reversed.

Once the scheduler completes all transfers for a request, it raises an interrupt and sets a tag status bit. The operating system receives the interrupt and completes the request by reading and then clearing the status bit using PIO operations.

The DMA controller manages Moneta's 2 GB/s full-duplex (4 GB/s total) channel to the host system. The DMA interleaves portions of bulk data transfers from Moneta to the host's memory (read requests) with DMA requests that retrieve data from host's memory controller (write requests). This results in good utilization of the bandwidth between the host and Moneta because bulk data transfers can occur in both directions simultaneously.

Memory controllers Each of Moneta's four memory controllers manages an independent bank of non-volatile storage. The controllers connect to the scheduler via the ring and provide a pair of 8 KB queues to buffer incoming and outgoing data. The memory array at each controller comprises four DIMM-like memory modules that present a 72 bit (64 + 8 for ECC) interface.

Like DRAM DIMMs, the memory modules perform accesses in parallel across multiple chips. Each DIMM contains four internal banks and two ranks, each with an internal row buffer. The banks and their row buffers are 8 KB wide, and the DIMM reads an entire row from the memory array into the row buffer. Once that data is in the row buffer, the memory controller can access it at 250 MHz DDR (500M transfers per second), so the peak bandwidth for a single controller is 4 GB/s.

The memory controller implements the start-gap wear-leveling and address randomization scheme [76] to evenly distribute wear across the memory it manages.

3.1.2 Implementing the Moneta prototype

We have implemented a Moneta prototype using the BEE3 FPGA prototyping system designed by Microsoft Research for use in the RAMP project [79]. The BEE3 system holds 64 GB of 667 MHz DDR2 DRAM under the control of four Xilinx Virtex 5 FPGAs, and it provides a PCIe link to the host system. The Moneta design runs at

250 MHz, and we use that clock speed for all of our results.

Moneta's architecture maps cleanly onto the BEE3. Each of the four FPGAs implement a memory controller, while one also implements the Moneta scheduler and the PCIe interface. The Moneta ring network runs over the FPGA-to-FGPA links that the BEE3 provides.

The design is very configurable. It can vary the effective number of memory controllers without reducing the amount of memory available, and it supports configurable buffer sizes in the scheduler and memory controllers.

Moneta memory controllers emulate PCM devices on top of DRAM using a modified version of the Xilinx Memory Interface Generator DDR2 controller. It adds latency between the read address strobe and column address strobe commands during reads and extends the precharge latency after a write. The controller can vary the apparent latencies for accesses to memory from 4 ns to over 500 μs. We use the values from [53] (48 ns and 150 ns for array reads and writes, respectively) to model PCM in this work, unless otherwise stated.

The width of memory arrays and the corresponding row buffers are important factors in the performance and energy efficiency of PCM memory [53]. The memory controller can vary the effective width of the arrays by defining a virtual row size and inserting latencies to model opening and closing rows of that size. The baseline configuration uses 8 KB rows.

3.2 Baseline Moneta performance

We use three metrics to characterize Moneta's performance. The first, shown in Figure 3.2, is a curve of sustained bandwidth for accesses to randomly selected locations over a range of access sizes. The second measures the average latency for a 4 KB access to a random location on the device. The final metric is the number of random access

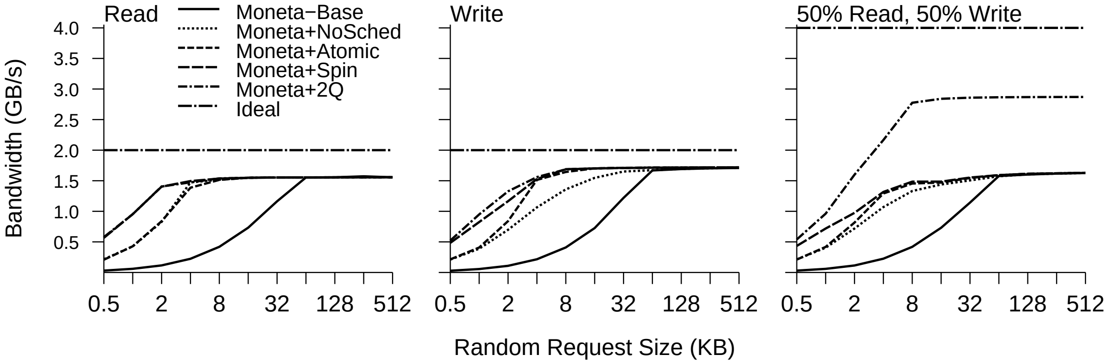

Figure 3.2. The PCIe link bandwidth (labeled "ideal") limits performance for large accesses, but for small accesses software overheads are the bottleneck for the baseline Moneta design (the solid black line). The other lines measure results for optimized version described in Section 3.3 and 3.4.

4 KB IO operations per second (IOPS) the device can perform with multiple threads.

Our test system is a two-socket, Core i7 Quad (a total of 8 cores) machine running at 2.26 GHz with 8 GB of physical DRAM and two 8 MB L2 caches. All measurements in this section are for a bare block device without a file system. We access Moneta via a low-level block driver and the standard Linux IO stack configured to use the No-op scheduler, since it provided the best performance. Each benchmark reports results for reads, writes, and a 50/50 combination of the two. We use XDD [109], a configurable IO performance benchmark for these measurements.

Figure 3.2 shows bandwidth curves for the baseline Moneta array and the optimized versions we discuss in future sections. The "ideal" curve shows the bandwidth of the 2 GB/s PCIe connection.

The baseline Moneta design delivers good performance, but there is room for improvement for accesses smaller than 64KB. For these, long access latency limits bandwidth, and much of this latency is due to software. Figure 3.3 and Table 3.1 breaks down the latency for 4 KB read and write accesses into 14 components. The data show that software accounts for 13 μs of the total 21 μs latency, or 62%. For comparison, the software overheads for issuing a request to the SSD- and disk-based RAID arrays described in Section 3.5 are just 17% and 1%, respectively. Although these data do not include a file system, other measurements show file system overhead adds an additional 4 to 5 μs of delay to each request.

These software overheads limit the number of IOPS that a single processor can issue to 47K, so it would take 10.5 processors to achieve Moneta's theoretical peak of 500K 4 KB read IOPS. As a result, IO-intensive applications running on Moneta with fewer than 11 processors will become CPU bound before they saturate the PCIe connection to Moneta.

These overheads demonstrate that software costs are the main limiter on Moneta performance. The next section focuses on minimizing these overheads.

3.3 Software Optimizations

3.3.1 IO scheduler

Linux IO schedulers sort and merge requests to reduce latencies and provide fair access to IO resources under load. Reordering requests can provide significant reductions in latency for devices with non-uniform access latencies such as disks, but for Moneta it just adds software overheads. Even the no-op scheduler, which simply passes requests directly to the driver without any scheduling, adds 2 μs to each request. This cost arises from context switches between the thread that requested the operation and the scheduler thread that actually performs the operation.

Moneta+NoSched bypasses the scheduler completely. It uses the thread that entered the kernel from user space to issue the request without a context switch. Removing the scheduler provides several ancillary benefits, as well. The no-op scheduler is single-threaded so it removes any parallelism in request handling. Under Moneta+NoSched

Table 3.1. The table breaks down baseline Moneta latency by component. Total latency is smaller than the sum of the components, due to overlap among components.

Label	Description	Baseline latency (μs)	
		Write	Read
OS/User	OS and userspace overhead	1.98	1.95
OS/User	Linux block queue and no-op scheduler	2.51	3.74
Schedule	Get request from queue and assign tag	0.44	0.51
Copy	Data copy into DMA buffer	0.24/KB	-
Issue	PIO command writes to Moneta	1.18	1.15
DMA	DMA from host to Moneta buffer	0.93/KB	-
Ring	Data from Moneta buffer to mem ctrl	0.28/KB	-
PCM	4 KB PCM memory access	4.39	5.18
Ring	Data from mem ctrl to Moneta buffer	-	0.43/KB
DMA	DMA from Moneta buffer to host	-	0.65/KB
Wait	Thread sleep during hw	11.8	12.3
Interrupt	Driver interrupt handler	1.10	1.08
Copy	Data copy from DMA buffer	-	0.27/KB
OS/User	OS return and userspace overhead	1.98	1.95
Hardware total for 4 KB (accounting for overlap)		8.2	8.0
Software total for 4 KB (accounting for overlap)		13.3	12.2
File system additional overhead		5.8	4.2

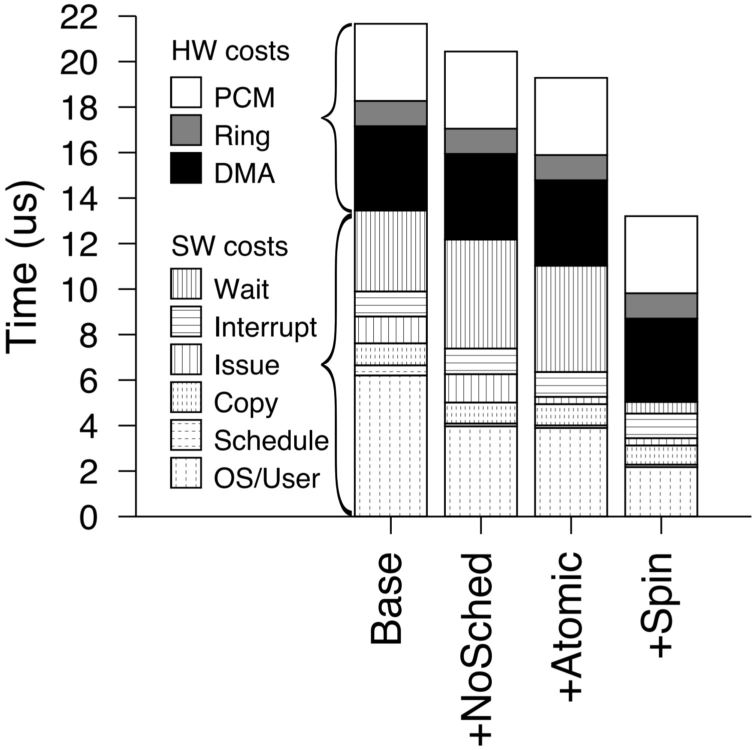

Figure 3.3. The figure shows that software optimizations reduce overall access latency by 9 μs between Moneta-Base and Moneta+Spin.

each request has a dedicated thread and those threads can issue and complete requests in parallel. Moneta+NoSched reduces per-request latency by 2 μs and increases peak bandwidth by 4× for 4 KB requests.

3.3.2 Issuing and completing IO requests

Moneta+NoSched allows some parallelism, but threads still contend for access to two locks that protect Moneta's hardware interface and shared data structures, limiting effective parallelism in the driver. The first lock protects the hardware interface during multi-word reads and writes to Moneta's PIO interface. The second lock protects shared data structures in the software.

To safely remove the lock that protects Moneta's hardware/software interface, we modify the interface so that issuing and completing a request requires a single, atomic PIO write or read. The baseline interface requires several PIO writes to issue a request. Moneta+Atomic reduces this to one by removing bits from the internal address and length fields to align them to 512-byte boundaries and completely removing the DMA address field. These changes allow a request to fit into 64 bits – 8 for the tag, 8 for the command, 16 for the length, and 32 for the internal address.

To specify the target address for DMA transfers, the Moneta+Atomic driver pre-allocates a DMA buffer for each tag and writes the host DRAM address of each buffer into a register in the hardware. The tag uniquely identifies which DMA buffer to use for each request.

A second change to Moneta's hardware/software interface allows multiple threads to process interrupts in parallel. When a request completes and Moneta raises an interrupt, the driver checks the status of all outstanding operations by reading the tag status register. In Moneta+NoSched the tag status register indicates whether each tag is busy or idle and the driver must read the register and then update it inside a critical section protected by a lock. In Moneta+Atomic the tag status bits indicate whether a request using that tag finished since the last read of the register, and reading the register clears it. Atomically reading and clearing the status register allows threads to service interrupts in parallel without the possibility of two threads trying to complete the same request.

The second lock protects shared data structures (e.g., the pool of available tags). To remove that lock, we reimplemented the tag pool data as a lock-free data structure, and allocate other structures on a per-tag basis.

By removing all the locks from the software stack, Moneta+Atomic reduces latency by 1 μs over Moneta+NoSched and increases bandwidth by 460 MB/s for 4 KB writes. The disproportionate gain in bandwidth versus latency results from increased concurrency.

3.3.3 Avoiding interrupts

Responding to the interrupt that signals the completion of an IO request requires a context switch to wake up the thread that issued the request. This process adds latency and limits performance, especially for small requests. Allowing the thread to spin in a busy-loop rather than sleeping removes the context switch and the associated latency. Moneta+Spin implements this optimization.

Spinning reduces latency by 6 μs, but it increases per-request CPU utilization. It also means that the number of thread-contexts available bounds the number of outstanding requests. Our data show that spinning only helps for write requests smaller than 4 KB, so the driver spins for those requests and sleeps for larger requests. Moneta+Spin improves bandwidth for 512-4096 byte requests by up to 250 MB/s.

3.3.4 Other overheads

The remaining overheads are from the system call/return and the copies to and from userspace. These provide protection for the kernel. Optimizing the system call interface is outside the scope of this chapter, but removing copies to and from userspace by directing DMA transfers into userspace buffers is a well-known optimization in high-performance drivers. For Moneta, however, this optimization is not profitable, because it requires sending a physical address to the hardware as the target for the DMA transfer. This would make issuing requests atomically impossible. Our measurements show that this hurts performance more than removing the copy to or from userspace helps. One solution would be to extend the processor's ISA to support 128 bit atomic writes. This would allow an atomic write to include the full DMA target address.

3.4 Tuning the Moneta hardware

With Moneta's optimized hardware interface and IO stack in place, we shift our focus to four aspects of the Moneta hardware – increasing simultaneous read/write bandwidth, increasing fairness between large and small transfers, adapting to memory technologies with longer latencies than PCM, and power consumption.

3.4.1 Read/Write bandwidth

Figure 3.2 shows that Moneta+Spin nearly saturates the PCIe link for read- and write-only workloads. For the mixed workload, performance is similar, but performance should be better since the PCIe link is full duplex.

Moneta+Spin uses a single hardware queue for read and write requests which prevents it from fully utilizing the PCIe link. Moneta+2Q solves this problem by providing separate queues for reads and writes and processing the queues in round-robin order. When reads and writes are both present, half of the requests to the DMA engine will be

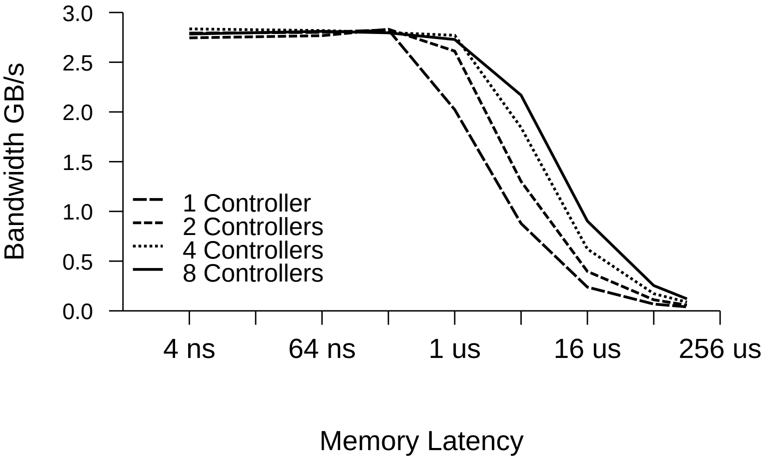

Figure 3.4. Additional controllers allow Moneta to hide longer memory latencies without sacrificing bandwidth. The curves measure bandwidth for 4 KB read and write accesses.

reads and half will be writes. The DMA engine interleaves the requests to transfer data in both directions simultaneously.

Figure 3.2 includes performance for Moneta+2Q. It increases mixed read/write performance by 75% and consumes 71% of the total PCIe bandwidth.

The new scheduler adds some complexity to the design and requires a change to how we implement IO barriers. IO barriers prevent the storage system from moving requests across the barrier, and some applications (e.g. file systems) use barriers to enforce ordering. The baseline Moneta design enforces this constraint by processing requests in order. Moneta+2Q's scheduler keeps track of outstanding barriers and synchronizes the read and write queue as needed to implement them. Requests that enter either queue after a barrier will wait for the barrier to complete before issuing.

3.4.2 Balancing bandwidth

Real-world application access patterns are more complex and varied than the workloads we have tested so far. Multiple applications will run concurrently placing different demands on Moneta simultaneously. Ideally, Moneta would prevent one access or access pattern from unduly degrading performance for other accesses.

The scheduler in Moneta+2Q processes the requests in each queue in order, so large requests can significantly delay other requests in the same queue. For a mixture of large and small requests, both sizes perform the same number of *accesses* per second, but bandwidth will be much smaller for the smaller accesses. For instance, if four threads issuing 4 KB reads run in parallel with four threads issuing 512 KB reads, the 4 KB threads realize just 16 MB/s, even though a 4 KB thread running in isolation can achieve 170 MB/s. The 512 KB threads receive a total 1.5 GB/s.

To reduce this imbalance, we modified the scheduler to service requests in the queue in round-robin order. The scheduler allocates a single buffer to the request at the front of the queue and then places the remainder of the request back on the queue before moving on to the next request. Round-robin queuing improves performance for the 4 KB threads in our example by 12×, and reduces large request bandwidth by 190 MB/s. Aggregate bandwidth remains the same.

The modified scheduler implements barriers by requiring that all in-progress requests complete before inserting any operations that follow the barrier into the scheduling queues.

3.4.3 Non-volatile memory latency

It is difficult to predict the latencies that non-volatile memories will eventually achieve, and it is possible that latencies will be longer than those we have modeled.

Figure 3.4 explores the impact of memory latency on Moneta's performance. It plots Moneta+Balance performance with memory latencies between 4 ns to 128 μs and with 1 to 8 memory controllers. The data show that adding parallelism in the form of memory controllers can completely hide latencies of up to 1 μs and that 4 μs memories would only degrade performance by 20% with eight controllers. The data also show that at 4 μs, doubling the number of controllers increases bandwidth by approximately 400 MB/s.

These results suggest that, for bandwidth-centric applications, it makes sense to optimize non-volatile memories for parallelism and interface bandwidth rather than latency. Furthermore, Moneta's memory controller only accesses a single bank of memory at a time. A more advanced controller could leverage inter-bank parallelism to further hide latency. Flash-based SSDs provide a useful case study: FusionIO's ioDrive, a state-of-the-art SSD, has many independent memory controllers to hide the long latencies of flash memory. In Section 3.5 we show that while a FusionIO drive can match Moneta's performance for large transfers, its latency is much higher.

3.4.4 Moneta power consumption

Non-volatile storage reduces power consumption dramatically compared to hard drives. To understand Moneta's power requirements we use the power model in Table 3.2. For the PCM array, we augment the power model in [53] to account for a power-down state for the memory devices. Active background power goes to clocking the PCM's internal row buffers and other peripheral circuitry when the chip is ready to receive commands. Each PCM chip dissipates idle background power when the clock is off, but the chip is still "on." The value here is for a commercially available PCM device [26]. We model the time to "wake up" from this state based on values in [26]. The model does not include power regulators or other on-board overheads.

The baseline Moneta array with four memory controllers, but excluding the memory itself, consumes a maximum of 3.19 W. For 8 KB transfers, the memory controller bandwidth limits memory power consumption to 0.6 W for writes, since transfer time over the DDR pins limits the frequency of array writes. Transfer time and power for reads is the same, but array power is lower. The PCIe link limits total array active memory power to 0.4 W (2 GB/s writes at 16.82 pJ/bit and 2 GB/s reads at 2.47 pJ/bit, plus row buffer power). Idle power for the array is also small – 0.13 W or 2 mW/GB. Total power for random 8 KB writes is 3.47 W, according to our model.

For small writes (e.g., 512 bytes), power consumption for the memory at a single controller could potentially reach 8.34 W since the transfer time is smaller and the controller can issue writes more frequently. For these accesses, peak controller bandwidth is 1.4 GB/s. Total Moneta power in this case could be as high as 11 W. As a result, efficiency for 512-byte writes in terms of MB/s/W drops by 85% compared to 8 KB writes.

The source of this inefficiency is a mismatch between access size and PCM row width. For small requests, the array reads or writes more bits than are necessary to satisfy the requests. Implementing selective writes [53] resolves this problem and limits per-controller power consumption to 0.6 W regardless of access size. Selective writes do not help with reads, however. For reads, peak per-controller bandwidth and power are 2.6 GB/s and 2.5 W, respectively, for 512-byte accesses.

An alternative to partial read and writes is to modify the size of the PCM devices' internal row buffer. Previous work [53] found this parameter to be an important factor in the efficiency of PCM memories meant to replace DRAM. For storage arrays, however, different optimizations are desirable because Moneta's spatial and temporal access patterns are different, and Moneta guarantees write durability.

Moneta's scheduler issues requests that are at least 512 bytes and usually several KB. These requests result in sequential reads and writes to memory that have much higher spatial locality than DRAM accesses that access a cache line at a time. In addition, since accesses often affect an entire row, there are few chances to exploit temporal locality. This means that, to optimize efficiency and performance, the row size and stripe size should be the same and they should be no smaller than the expected transfer size.

Table 3.2. The component values for the power model come from datasheets, Cacti [99], [60], and the PCM model in [53]

Component	Idle	Peak
Scheduler & DMA [45]	0.3 W	1.3 W
Ring [60]	0.03 W	0.06 W
Scheduler buffers [99]	1.26 mW	4.37 mW
Memory controller [60]	0.24 W	0.34 W
Mem. ctrl. buffers [99]	1.26 mW	4.37 mW
PCIe[45]	0.12 W	0.4 W
PCM write [53]		16.82 pJ/bit
PCM read [53]		2.47 pJ/bit
PCM buffer write [53]		1.02 pJ/bit
PCM buffer read [53]		0.93 pJ/bit
PCM background [53, 70]	264 μW/die	20 μW/bit

Finally, Moneta's durability requirements preclude write coalescing, a useful optimization for PCM-based main memories [53]. Coalescing requires keeping the rows open to opportunistically merge writes, but durability requires closing rows after accesses complete to ensure that the data resides in non-volatile storage rather than the volatile buffers.

3.5 Evaluation

This section compares Moneta's performance to other storage technologies using both microbenchmarks and full-fledged applications. We compare Moneta to three other high-performance storage technologies (Table 3.3) – a RAID array of disks, a RAID array of flash-based SSDs, and an 80GB FusionIO flash-based PCIe card. We also estimate performance for Moneta with a 4x PCIe link that matches the peak bandwidth of the other devices.

We tuned the performance of each system using the IO tuning facilities available for Linux, the RAID controller, and the FusionIO card. Each device uses the best-performing of Linux's IO schedulers: For RAID-disk and RAID-SSD the no-op scheduler

Table 3.3. We compare Moneta to a variety of existing disk and SSD technologies. Bus bandwidth is the peak bandwidth each device's interface allows.

Name	Bus	Description
RAID-Disk	PCIe 1.1 ×4 1 GB/s	4×1TB hard drives. RAID-0 PCIe controller.
RAID-SSD	SATA II ×4 1.1 GB/s	4×32GB X-25E SSDs. RAID-0 Software RAID.
FusionIO	PCIe 1.1 ×4 1 GB/s	Fusion-IO 80GB PCIe SSD
Moneta-4x	PCIe 1.1 ×4 1 GB/s	See text.
Moneta-8x	PCIe 1.1 ×8 2 GB/s	See text.

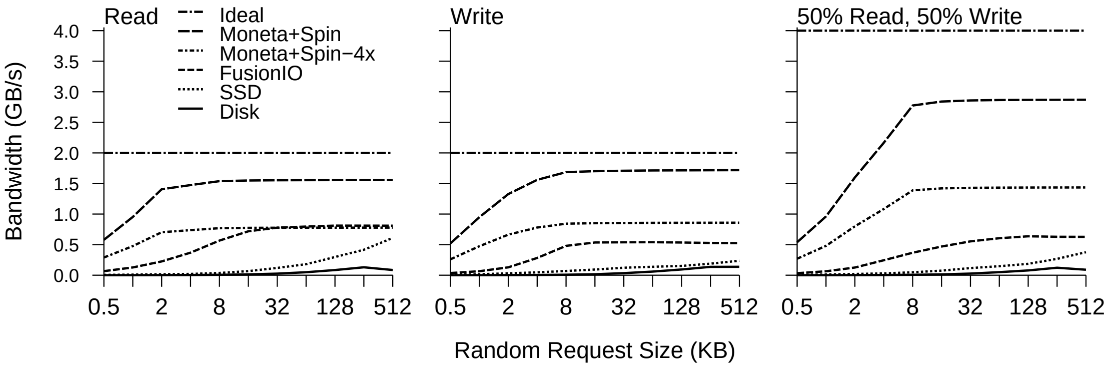

Figure 3.5. Small request sizes measure random access performance, while larger transfers measure sequential accesses. Moneta+Spin-4x conservatively models a 1GB/s PCIe connection and matches or outperforms the other storage arrays.

Table 3.4. Moneta's combination of fast storage and optimized software stack allow it to perform 1.1M 512 byte IOPS and 370K 4 KB IOPS. Software optimizations also reduces the number of instructions per IO operations by between 38 and 47%.

	4KB IOPS		512 IOPS		Instructions per 4 KB IOP	
	Read	Write	Read	Write	Read	Write
RAID-Disk	863	1,070	869	1,071	32,042	34,071
RAID-SSD	5,256	11,714	11,135	17,007	31,251	32,540
FusionIO	92,225	70,579	132,826	71,338	49,369	60,147
Moneta+Spin	368,534	389,859	1,156,779	1,043,793	17,364	20,744

performs best. The FusionIO card uses a custom driver which bypasses the Linux IO scheduler just as Moneta's does. We use software RAID for the SSD rather than hardware because our measurements and previous studies [38] show better performance for that configuration.

3.5.1 Microbenchmarks

Figure 3.5 and Table 3.4 contain the results of the comparison. For bandwidth, Moneta+Spin outperforms the other arrays by a wide margin, although for large transfer sizes most of this benefit is due to its faster PCIe link. For mixed reads and writes, Moneta delivers between 8.9× (vs. FusionIO) and 616× (vs. disk) more bandwidth. For Moneta+Spin-4x, the gains are smaller (4.4-308×). For mixed 512 KB read and write accesses Moneta is 32× faster than disk and 4.5× faster than FusionIO. The bandwidth gains relative to the SSDs vary less widely – between 7.6 and 89× for Moneta and between 3.8 and 44× for Moneta+Spin-4x.

Moneta also delivers large gains in terms of IOPS. Moneta sustains between 364 and 616× more 4 KB IOPS than disk and between 4 and 8.9× more than FusionIO. The gains for small transfers are much larger: Moneta achieves 1.1M 0.5 KB IOPS, or 10× the value for FusionIO.

Table 3.4 also shows the number of instructions each device requires to perform a 4 KB read or write request. Moneta's optimized driver requires between 38 and 47% fewer instructions than RAID-Disk and RAID-SSD. The reduction relative to the baseline Moneta software stack is similar. As a result, it takes five processors (rather than eleven) to fully utilize the Moneta array. FusionIO's driver requires between 1.4 and 1.8× more instructions than the RAID arrays because it caches device meta-data to increase performance.

3.5.2 Applications

Moneta's optimized hardware interface and software stack eliminates most of the operating system overheads that limited the performance for raw device access. Here, we explore how that performance translates to the application level and highlight where further software optimizations may be useful.

Table 3.5 describes the workloads we use in this evaluation. They fall into three categories that represent potential uses for Moneta: IO benchmarks that use typical system calls to access storage, database applications that implement transaction systems with strong durability guarantees, and paging applications that use storage to back virtual memory. Whenever practical and possible, we use direct IO to bypass the system buffer cache for Moneta and FusionIO, since it improves performance. Adding the ability to apply this setting without modifying application source code would be a useful optimization.

Table 3.6 shows the performance of the storage arrays and the speedup of Moneta over each array. All data use XFS [48] as the underlying file system because it efficiently supports parallel file access.

The performance of the IO benchmarks improves significantly running on Moneta: On average, Moneta is 9.48× faster than RAID-disk, 2.84× faster than RAID-SSD, and 2.21× faster than FusionIO. However, the performance of XDD-Random on Moneta is

Table 3.5. We use a total of fifteen benchmarks and workloads to compare Moneta to other storage technologies. Workloads vary from simple microbenchmarks to larger database applications.

Name	Data footprint	Description
IO benchmarks		
XDD-Sequential-Raw	55 GB	4MB sequential reads/writes from 16 threads through the raw block device
XDD-Sequential-FS	55 GB	4MB sequential reads/writes from 16 threads through the file system
XDD-Random-Raw	55 GB	4KB random reads/writes from 16 threads through the raw block device
XDD-Random-FS	55 GB	4KB random reads/writes from 16 threads through the file system
Postmark	≈0.3-0.5 GB	Models an email server
Build	0.5 GB	Compilation of the Linux 2.6 kernel
Database applications		
PTFDB	50 GB	Palomar Transient Factory database realtime transient sky survey queries
OLTP	8 GB	Transaction processing using Sysbench running on a MySQL database
Berkeley-DB Btree	4 GB	Transactional updates to a B+tree key/value store
Berkeley-DB Hashtable	4 GB	Transactional updates to a hash table key/value store
Paging applications		
BT	11.3 GB	Computational fluid dynamics simulation
IS	34.7 GB	Integer sort with the bucket sort algorithm
LU	9.4 GB	LU matrix decomposition
SP	11.9 GB	Simulated CFD code solves Scalar-Pentadiagonal bands of linear equations
UA	7.6 GB	Solves a heat transfer problem on an unstructured, adaptive grid

Table 3.6. We run our fifteen benchmarks and applications on the RAID-Disk, RAID-SSD, FusionIO, and Moneta storage arrays, and we show the speedup of Moneta over the other devices.

Workload	Raw Performance				Speedup of Moneta vs.		
	RAID-Disk	RAID-SSD	FusionIO	Moneta	RAID-Disk	RAID-SSD	FusionIO
IO benchmarks							
Postmark	4,080.0 s	46.8 s	36.7 s	27.4 s	149.00×	1.71×	1.34×
Build	80.9 s	36.9 s	37.6 s	36.9 s	2.19×	1.00×	1.02×
XDD-Sequential-Raw	244.0 MB/s	569.0 MB/s	642.0 MB/s	2,932.0 MB/s	12.01×	5.15×	4.57×
XDD-Sequential-FS	203.0 MB/s	564.0 MB/s	641.0 MB/s	2,773.0 MB/s	13.70×	4.92×	4.35×
XDD-Random-Raw	1.8 MB/s	32.0 MB/s	142.0 MB/s	1,753.0 MB/s	973.89×	54.78×	12.35×
XDD-Random-FS	3.3 MB/s	30.0 MB/s	118.0 MB/s	261.0 MB/s	80.40×	8.70×	2.21×
Harmonic mean					9.48×	2.84×	2.21×
Database applications							
PTFDB	68.1 s	5.8 s	2.3 s	2.1 s	32.60×	2.75×	1.11×
OLTP	304.0 tx/s	338.0 tx/s	665.0 tx/s	799.0 tx/s	2.62×	2.36×	1.20×
Berkeley-DB BTree	253.0 tx/s	6,071.0 tx/s	5,975.0 tx/s	14,884.0 tx/s	58.80×	2.45×	2.49×
Berkeley-DB HashTable	191.0 tx/s	4,355.0 tx/s	6,200.0 tx/s	10,980.0 tx/s	57.50×	2.52×	1.77×
Harmonic mean					8.95×	2.51×	1.48×
Paging applications							
BT	84.8 MIPS	1,461.0 MIPS	706.0 MIPS	2,785.0 MIPS	32.90×	1.90×	3.94×
IS	176.0	252.0	252.0	351.0	2.00×	1.39×	1.39×
LU	59.1	1,864.0	1,093.0	4,298.0	72.70×	2.31×	3.93×
SP	87.0	345.0	225.0	704.0	8.08×	2.04×	3.13×
UA	57.2	3,775.0	445.0	3,992.0	69.80×	1.06×	8.97×
Harmonic mean					7.33×	1.61×	3.01×

much lower when reads and writes go through the file system instead of going directly to the raw block device. This suggests that further optimizations to the file system may provide benefits for Moneta, just as removing device driver overheads did. Performance on Build is especially disappointing, but this is in part because there is no way to disable the file buffer cache for that workload.

The results for database applications also highlight the need for further software optimizations. Databases use complex buffer managers to reduce the cost of IO, but our experience with Moneta to date suggests reevaluating those optimizations. The results for Berkeley DB bear this out: Berkeley DB is simpler than the PostgreSQL and MySQL databases that OLTP and PTFDB use, and that simplicity may contribute to Moneta's larger benefit for Berkeley DB.

For the paging applications, Moneta is on average 7.33×, 1.61×, and 3.01× faster than RAID-Disk, RAID-SSD, and FusionIO, respectively. For paging applications, Moneta is on average 2.75× faster than the other storage arrays. Not surprisingly, the impact of paging to storage depends on the memory requirements of the program, and this explains the large variation across the five applications. Comparing the performance of paging as opposed to running these applications directly in DRAM shows that Moneta reduces performance by only 5.90× as opposed to 14.7× for FusionIO, 13.3× for RAID-SSD, and 83.0× for RAID-Disk. If additional hardware and software optimizations could reduces this overhead, Moneta-like storage devices could become a viable option for increasing effective memory capacity.

3.6 Related work

Software optimizations and scheduling techniques for IO operations for disks and other technologies has been the subject of intensive research for many years [67, 98, 108, 89, 97, 92, 69]. The main challenge in disk scheduling is minimizing the impact of the long rotational and seek time delays. Since Moneta does not suffer from these delays, the scheduling problem focuses on exploiting parallelism within the array and minimizing hardware and software overheads.

Recently, scheduling for flash-based solid-state drives has received a great deal of attention [49, 24, 14]. These schedulers focus on reducing write overheads, software overheads, and exploiting parallelism within the drive. The work in [90] explores similar driver optimizations for a PCIe-attached, flash-based SSD (although they do not modify the hardware interface) and finds that carefully-tuned software scheduling is useful in that context. Our results found that any additional software overheads hurt Moneta's performance, and that scheduling in hardware is more profitable.

In the last decade, there have also been several proposals for software scheduling policies for MEMS-based storage arrays [55, 5, 87, 100, 23]. Other researchers have characterized SSDs [25, 21, 15] and evaluated their usefulness on a range of applications [68, 57, 16, 4]. Our results provide a first step in this direction for faster non-volatile memories.

Researchers have developed a range of emulation, simulation, and prototyping infrastructures for storage systems. Most of these that target disks are software-based [31, 25, 83, 37, 50]. Recently, several groups have built hardware emulation systems for exploring flash-based SSD designs: The work in [20, 56, 52] implements flash memory controllers in one or more FPGAs and attach them to real flash devices. Moneta provides a similar capability for fast non-volatile memories, but it emulates the memories using several modifications in the memory controller (described in Appendix A). The work in [20] uses the same BEE3 platform that Moneta uses.

3.7 Summary

We have presented Moneta, a storage array architecture for advanced non-volatile memories. A series of software and hardware interface optimizations significantly improves Moneta's performance. Our exploration of Moneta designs shows that optimizing PCM for storage applications requires a different set of trade-offs than optimizing it as a main memory replacement. In particular, memory array latency is less critical for storage applications if sufficient parallelism is available, and durability requirements prevent some optimizations.

Optimizations to Moneta's hardware and software reduce software overheads by 62% for 4 KB operations, and enable sustained performance of 1.1M 512-byte IOPS and 541K 4 KB IOPS with a maximum sustained bandwidth of 2.8 GB/s. Moneta's optimized IO stack completes a single 512-byte IOP in 9 μs. Moneta speeds up a range of file system, paging, and database workloads by up to 8.7× compared to a state-of-the-art flash-based SSD with harmonic mean of 2.1×, while consuming a maximum power of 3.2 W.

Acknowledgements

This chapter contains material from "Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing", by Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson, which appears in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '10). The dissertation author was the primary investigator and author of this paper.

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, which appears in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (MICRO ’43). The dissertation author was the primary 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.

Chapter 4

User Space Access

The previous chapter examined Moneta's kernel software stack and showed that optimizing the existing IO stack and tuning the hardware/software interface can reduce software overheads by up to 62% and increase sustained bandwidth for small accesses by up to 19×. However, even with these reduced overheads, IO processing places large demands on the system's compute resources – sustaining peak performance on Moneta for 4 KB requests requires the dedicated attention of 9 Nehalem thread contexts. Entering the kernel, performing file system checks, and returning to user space account for 30% (8 μs) of the latency of a 4 KB request. Together those costs reduce sustained throughput by 85%. However, simply removing those layers is not possible because they provide essential management and protection mechanisms.

This chapter describes extensions to Moneta that remove these costs by transparently bypassing the operating and file systems while preserving their management and protection functions. The extensions provide each process with a private interface, or *channel*, to Moneta. Unlike other systems that virtualize an entire device (e.g., a graphics card or network card), Moneta's channels are virtual *interfaces* to a single device. Each process uses its channel to access Moneta directly, without interacting with the operating system for most accesses. Hardware permission verification replaces permission checks in the operating system, preserving all of the protection guarantees the operating system normally provides.

To utilize channel-based IO, unmodified applications link with an untrusted user space library that intercepts IO system calls and performs the operations directly. The library works with a trusted driver and a slightly modified file system to extract file protection information and transfer it to Moneta. The library presents a standard POSIX interface.

We refer to the new system as *Moneta Direct* (*Moneta-D*) throughout the remainder of this dissertation. Moneta-D's unique features eliminate file system overheads and restructure the operating system storage stack to efficiently support direct, user space access to fast non-volatile storage arrays:

- Moneta-D removes trusted code and the associated overheads from most accesses. Only requests that affect meta-data need to enter the operating system.

- Moneta-D provides a fast hardware mechanism for enforcing the operating and file system's data protection and allocation policy.

- Moneta-D trades off between CPU overhead and performance by using different interrupt forwarding strategies depending on access size and application requirements.

- Moneta-D provides an asynchronous software interface, allowing applications to leverage its inherently asynchronous hardware interface to increase performance and CPU efficiency.

We evaluate options for key design decisions in Moneta-D and measure its performance under a collection of database workloads and direct file access benchmarks. Our results show that Moneta-D improves performance for simple database operations by between 2.6× and 5.7×. For full SQL server workloads, performance improves by between 1.1× and 2.0×. For file access benchmarks, our results show that Moneta-D can reduce the latency for a 512 byte read operation by 64% (to 4.1 μs) relative to the original Moneta design. The reduction in latency leads to a 14.8× increase in bandwidth for 512 byte requests in the presence of a file system, allowing the storage array to sustain up to 1.8 million 512 byte IO operations per second. With a single thread, Moneta-D's asynchronous IO interface improves performance by 5.5× for 4 KB accesses compared to synchronous IO. Asynchronous IO also improves efficiency by up to 2.8×, reducing CPU utilization and saving power.

The remainder of this chapter is organized as follows. Section 4.1 describes the baseline Moneta storage array and provides an overview of the changes required to virtualize it. Section 4.2 places our work in the context of other research efforts. Section 4.3 describes Moneta-D's architectural support for virtualization. Section 4.4 evaluates the system. Finally, Section 4.5 concludes.

4.1 System overview

Moneta-D's goal is to remove operating system and file system overheads from accesses while maintaining the strong protection guarantees that these software layers provide. The resulting system should be scalable in that many applications should be able to access Moneta-D concurrently without adversely affecting performance. Furthermore, it should not be necessary to modify the applications to take advantage of Moneta-D.

Figures 4.1(a) and (b) summarize the changes Moneta-D makes to the hardware and software components of the original Moneta system. In Moneta (Figure 4.1(a)), all interactions with the hardware occur via the operating system and file system. Together, they set the policy for sharing the device and protecting the data it contains. They also enforce that policy by performing checks on each access. Performing those checks requires the file system to be trusted code. The driver is trusted since it accesses a shared,

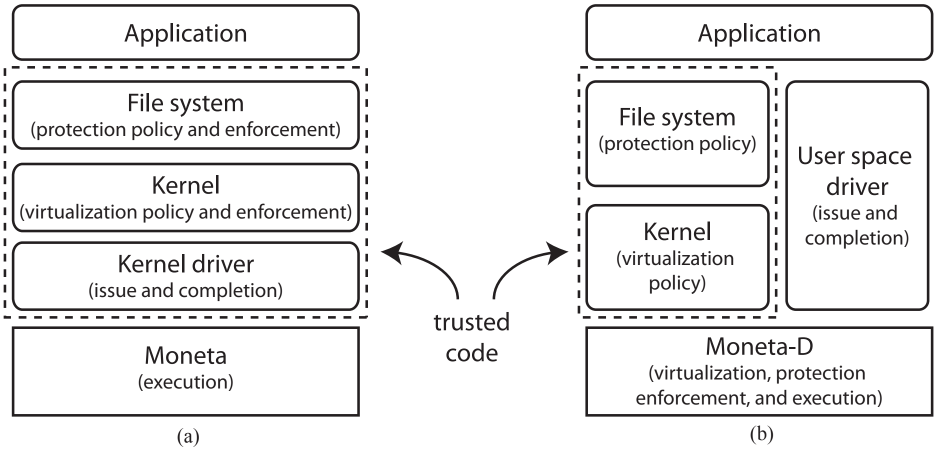

Figure 4.1. The operating and file systems protect storage devices like Moneta (a) by requiring a system call on every access. In Moneta-D (b), the OS and file system are only responsible for setting protection and sharing policy. Moneta-D enforces that policy in hardware, eliminating the need for most system calls and file system accesses. A virtualized hardware interface allows for an untrusted driver.

non-virtualized hardware resource.

Figure 4.1(b) shows the revised organization that Moneta-D uses. The kernel and the file system remain responsible for making policy decisions that control access to data, but the Moneta-D hardware enforces that policy. The hardware exposes a set of virtual *channels* that each provide a single process with access to storage. The kernel manages these channels, assigns them to processes, and maintains protection information associated with each channel. Since the hardware enforces protection and each process has a private channel, there is no need for a privileged driver. Instead, applications access their channels via an untrusted driver library, avoiding system call overheads.

Our intent is that this new architecture be the default mechanism for file access rather than a specialized interface for high-performance applications. To make it feasible for all applications running on a system to use the interface, Moneta-D supports a large number of virtual channels. This decision has forced us to minimize the cost of virtualization.

Below, we describe the channel interface, the user space driver library, and discuss the interactions between our system and the file system.

4.1.1 Channels

A channel is a virtual interface to the storage array. Each channel provides all the facilities necessary for a process to access data in the array and for the kernel to restrict access to only files that the process has successfully opened.

A channel has two interfaces, a privileged interface for configuration by the kernel and an unprivileged interface for application access. The privileged interface comprises a set of control registers that let the kernel manage the channel and install permission information. The unprivileged interface has three components: 1) a set of user registers that the user space library uses to access array data, 2) a set of tags that distinguish between outstanding requests on the channel, and 3) a DMA buffer.

Below, we describe how a process and the kernel use their respective interfaces to initialize a channel, access data, and manage permission information. Section 4.3 describes these mechanisms in more detail.

Channel initialization The user space driver library initializes a channel by opening the storage device's file in `/dev/` and `mmap()`ing several pages into its address space. Mapping these pages allocates a channel for the process and grants it access to the hardware and shared-memory software interfaces. The first mapped page contains the user registers that the process will use to communicate with the hardware. The next pages provide communication with the kernel via shared memory. The remaining pages make up the channel's DMA buffer. Initially, the channel does not have permission to access any of the data in Moneta-D.

Managing permissions To gain access to data in Moneta-D, the user space library issues a system call that takes a file descriptor and an offset in the file. The system call returns a description of the file system extent (i.e., the range of physical bytes in Moneta-D) containing that offset. The process uses this information to populate a user space table that maps file offsets onto physical extents. If the process does not have access to that data, the system call returns an error.

The system call also installs the permission record for the extent in the process's channel. Moneta-D's permission record storage is finite, so installing one permission record may require the kernel to evict another. This also means that the process may issue a request for data that it should be able to access and have the request fail. In this case, the process re-issues the system call to re-install the permission record and retries the request.

Issuing and completing commands Once the process has installed a permission record, it can start making requests. To initiate a command, the process writes a 64-bit command word to the channel's memory-mapped command register. The command encodes the operation to perform (read or write), the portion of the DMA buffer to use, the physical location in Moneta-D to access, and a tag to differentiate between requests. After issuing the command, the thread waits for the command to complete.

When Moneta-D receives the command word, it checks the hardware permission table to determine whether the channel has permission to access the location. If it does, it performs the command and signals its completion. In Section 4.3 we describe several schemes for notifying the thread when a command completes.

4.1.2 The user space driver

The user space library for accessing Moneta-D performs the low-level driver functions including tag management, extent lookup, and command retry. The library transparently replaces the standard library calls for accessing files using LD_PRELOAD. Dynamically linked applications do not require any modification or recompilation. When the program `open()`s a file on Moneta-D, the library allocates a channel if necessary and then handles all future accesses to that file. The library forwards operations on all other files to the normal `libc` functions.

The POSIX compatibility layer implements POSIX calls (e.g. `read()`, `write()`, and `seek()`) calling Moneta-D specific read and write functions. The layer also tracks file descriptor manipulation functions (e.g. `dup()`, `dup2()`, and `close()`) to track per-file descriptor state (e.g. the file pointer's position) and file descriptor aliasing relationships.

Other, Non-POSIX interfaces are also possible. Moneta-D's hardware interface is inherently asynchronous, so a high-performance asynchronous IO library is a natural fit. In addition, since the channel's DMA buffers reside in the process's address space, an optimized application could avoid copying data to and from the DMA buffer and operate on the data in place instead. We explore both these options in Section 4.3.

4.1.3 The file system

Moneta-D changes the way applications interact with the file system to increase performance. These changes require minor modifications in the file system to support moving protection checking into hardware. They also introduce some challenges to maintaining existing functionality and consistency in the file system.

The only change required to the file system is the addition of a function to extract extent information. We implemented this change in XFS [48] and found it to be straightforward, even though XFS is a very sophisticated file system. The single 30-line function accesses and transfers file extent meta-data into Moneta-D's data structures. We expect that adding support to other file systems would also be relatively easy.

All meta-data updates and accesses use the conventional operating system interface. This requirement creates problems when the driver uses the kernel to increase a file's size and then accesses the resulting data directly. When it extends a file (or fills a hole in a sparse file), XFS writes zeros into the kernel's file buffer cache. Although the operating system writes out these dirty blocks after a short period of time, the user space library accesses the newly-allocated blocks as soon as the system call returns and that access will not ”see” the contents of the buffer cache. If the application updates the new pages in Moneta-D before the operating system flushes the cached copy, a race will occur and the flushed pages will overwrite the application's changes. To avoid this problem, we flush all blocks associated with a file whenever we fill a hole or extend a file. After the first access to the file, this is usually a fast operation because the buffer cache will be empty.

Guaranteeing consistency while accessing files concurrently through Moneta-D and via the operating system remains a challenge, because of potential inconsistencies caused by the kernel's buffer cache. One solution is to detect files that applications have opened a file using both interfaces and force the application using the user space interface to switch to the system-call based interface. The library could do this without that application's knowledge. Alternatively, disabling the buffer cache for files residing on Moneta-D would also resolve the problem. Our system does not yet implement either option.

Moneta-D's virtual interface also supports arbitrarily sized and aligned reads and writes, eliminating the need to use read-modify-write operations to implement unaligned accesses.

4.2 Related Work

Our extensions to Moneta touch on questions of virtualization, fast protection and translation, and light-weight user space IO. Below we describe related work in each of these areas and how the system we describe differs from and extends the current state of the art.

4.2.1 Virtualization

Moneta-D differs from other efforts in virtualizing high-speed IO devices in that it provides virtual *interfaces* to the device rather than logically separate virtual devices. In our system, there is a single logical SSD and a single file system to manage it. However, many client applications can access the hardware directly. Creating multiple, independent virtual disks or multiple, independent virtual network interfaces for multiple virtual machines is a simpler problem because the virtual machine monitor can statically partition the device's resources across the virtual machines.

Previous work in high-speed networking [102, 10, 73] explores the idea of virtualizing network interfaces and allowing direct access to the interface from user space. DART [73] implements network interface virtualization while also supporting offloading of some packet processing onto the network card for additional performance enhancements. Our work implements similar hardware interface virtualization for storage accesses while enabling file systems protection checking in hardware.

Many projects have developed techniques to make whole-device virtualization more efficient [95, 64, 82, 66, 107], particularly for graphics cards and high-performance message-passing interconnects such as Infiniband. Virtualization techniques for GPUs are most similar to our work. They provide several "rendering contexts" that correspond to an application window or virtual machine [22]. A user space library (e.g., OpenGL) requests a context from the kernel, and the kernel provides it a set of buffers and control registers it can use to transfer data to and from the card without OS involvement. Some Infiniband cards [36] also provide per-application (or per-virtual machine) channels and split the interface into trusted and untrusted components. The work in [62] has explored how to expose these channels directly to applications running inside virtual machines. However, neither of these applications requires the hardware to maintain fine-grain permission data as Moneta-D does.

The concurrent, direct network access (CDNA) model [107] is also similar, but applies to virtual machines. In this model, the network card provides multiple independent sets of queues for network traffic, and the VMM allows each virtual machine to access one of them directly. On an interrupt, the OS checks a register to determine which queues need servicing and forwards a virtual interrupt to the correct VMs.

Recent revisions of the PCIe standard include IO virtualization (IOV) [74] to support virtual machine monitors. PCIe IOV allows a single PCIe device to appear as several, independent virtual devices. The work in [110] describes a software-only approach to virtualizing devices that do not support virtualization, assuming the devices satisfies certain constraints. In both cases the support is generic, so it cannot provide the per-channel protection checks that Moneta-D requires. Some researchers have also found the PCIe approach to be inflexible in the types of virtualized devices it can support [58].

The work in [84] and [78] present new IO architectures with virtualization as the driving concern. In [84] researchers propose a unified interface to several of the techniques described above as well as extensions to improve flexibility. [78] proposes a general approach to self-virtualizing IO devices that offloads many aspects of virtualization to a processor core embedded in the IO device.

4.2.2 User space IO

Efficiently initiating and completing IO requests from user space has received some attention in the high-speed networking and message passing communities. In almost all cases, the VMs issue requests via stores to PIO registers, and the VMM is responsible for delivering virtual interrupts to the VMs. We describe two alternative approaches below.

Prior work [102, 10, 8] proposed supporting user space IO and initiating DMA transfers from user space without kernel intervention. SHRIMP [8] proposes user space DMA through simple load and store operations, but requires changes to the CPU and DMA engine to detect and initiate transfers. Our work requires no changes to the CPU or chipset.

The work in [86, 85] proposes architectural support for issuing multi-word PIO commands atomically. In effect, it implements a simple form of bounded transactional memory. Moneta-D would be more flexible if our processors provided such support. The same work also suggests adding a TLB to the PCIe controller to allow the process to specify DMA targets using virtual addresses. The PCIe IOV extensions mentioned above provide similar functions. The combination of multi-word atomic PIO writes and the DMA TLB would eliminate the need for a dedicated DMA buffer and make it possible to provide a zero-copy interface on top of Moneta-D that was POSIX-compatible.

The same work also proposes hardware support for delivering interrupts to user space. The device would populate a user space buffer with the results of the IO operation, and then transmit data to the CPU describing which process should receive the interrupt. The OS would then asynchronously execute a user-specified handler. Moneta-D would benefit from this type of support as well, and one of the request completion techniques we examine is similar in spirit. More recently, researchers have proposed dedicating an entire core to polling IO device status and delivering notifications to virtual machines through memory [61]. The driver for recent PCIe-attached flash-based SSDs from Virident dedicates one processor core solely to interrupt handling.

Several papers have argued against user space IO [63]. They posit that efficient kernel-level implementations can be as fast as user-level ones and that the kernel should be the global system resource controller. However, our work and prior work [114] have found that user-level IO can provide significant benefit without significantly increasing complexity for application developers. Our work maintains the kernel as the global policy controller — only policy enforcement takes place in hardware.

4.2.3 Protection and translation

Moneta-D removes file system latency by copying permission information into hardware and caching the physical layout of data in user space. Some approaches to distributed, networked storage use similar ideas. The latest version of the network file system (NFS) incorporates the pNFS [39] extension that keeps the main NFS server from becoming a bottleneck in cluster-based NFS installations. Under pNFS, an NFS server manages storage spread across multiple storage nodes. When a client requests access to a file, it receives a map that describes the layout of the data on the storage nodes. Further requests go directly to the storage nodes. NASD [34] is similar in that a central server delegates access rights to clients. However, it uses intelligent drives rather than separate storage servers to provide access to data. NASD uses cryptographic capabilities to grant clients access to specific data.

Modern processors provide hardware support for translation and protection (the TLB) and for servicing TLB misses (the page table walker) in order to reduce both translation and miss costs. Supporting multiple file systems, many channels, and large files requires Moneta-D to take a different approach. Our SSD provides hardware support for protection only. Translation must occur on a per-file basis (since “addresses” are file offsets), and hardware translation would require Moneta-D to track per-file state rather than per-channel state.

Rather than addressing physical blocks in a storage device, object-based storage systems [1, 35, 105], store objects addressed by name. They provide a layer of abstraction mapping between object names and physical storage in the device. Moneta-D performs a similar mapping in its user space library, mapping between file descriptor and offset. Shifting these translations into the hardware has several drawbacks for a system like Moneta-D. First, the file system would require significant alterations, breaking the generic support that Moneta-D currently enables. Second, performing the translations directly in hardware could limit Moneta-D’s performance if the lookups take more than a few hundred nanoseconds. Finally, dedicated DRAM in Moneta-D for storing lookup information might be better located in the host system where it could be repurposed for other uses when not needed for translations.

4.3 Moneta-D Implementation

This section describes the changes to the Moneta [12] hardware and software that comprise Moneta-D. The baseline Moneta system implements a highly optimized SSD architecture targeting advanced non-volatile memories. The Moneta-D modifications enable the hardware and software to work together to virtualize the control registers and tags, efficiently manage permission information, and deliver IO completions to user space. We discuss each of these in turn. Section 4.4 evaluates their impact on performance.

Table 4.1. There are three interfaces that control Moneta-D: The kernel registers that the kernel uses to configure channels, the per-channel user registers that applications use, and the per-channel mapped memory region shared between the kernel and the application to maintain channel state.

		R/W			
	Name	Kernel	User	HW	Description
Kernel global registers	ChannelStatus	R	-	W	Read and clear channel status and error bits
	ErrorQueue	R	-	W	Read and pop one error from the error queue
User per-channel registers	Command	W	W	R	Issue a command to the device.
	TagStatusRegister	R	R	W	Read and clear tag completion bits and error flag.
Per-channel mapped memory	TagStatusTable	W	R/W	W	Tracks completion status of outstanding requests.
	CompletionCount	W	R	-	Count of completed requests on each channel.
	DMABuffer	-	R/W	R/W	Pinned DMA buffer for data transfers.

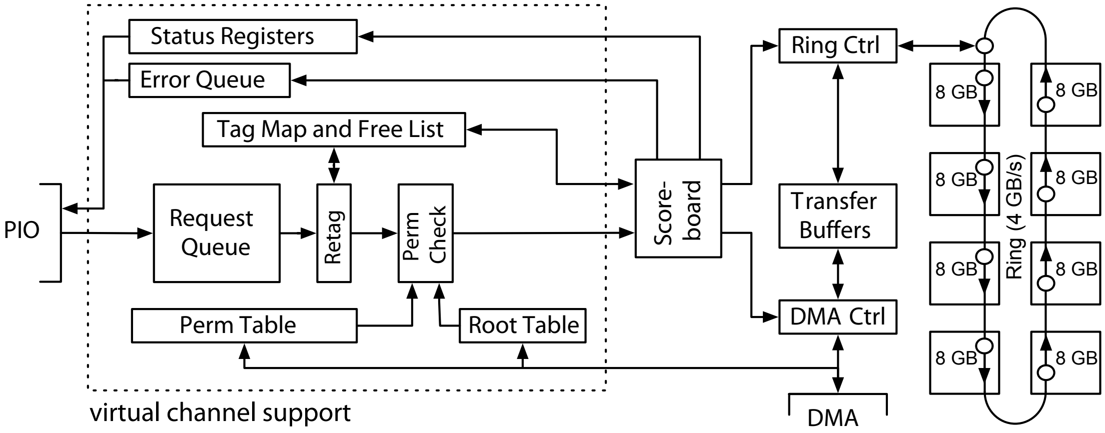

Figure 4.2. Components inside the dashed box provide support for virtualization and are the focus of this work. The other components (at right) execute storage access commands.

4.3.1 The baseline Moneta hardware

The right side of Figure 4.2 (outside the dotted box) shows the architecture of the baseline array. It spreads 64 GB of storage across eight memory controllers connected via a high-bandwidth ring. An 8-lane PCIe 1.1 interface provides a 2 GB/s full-duplex connection (4 GB/s total) to the host system. The baseline design supports 64 concurrent, outstanding requests with unique tags identifying each. The prototype runs at 250 MHz on a BEE3 FPGA prototyping system [7].

The baseline Moneta array emulates advanced non-volatile memories using DRAM and modified memory controllers that insert delays to model longer read and write times. We model phase change memory (PCM) in this work and use the latencies from [53] — 48 ns and 150 ns for array reads and writes, respectively. The array uses start-gap wear leveling [77] to distribute wear across the PCM and maximize lifetime.

The baseline Moneta design includes extensive hardware and software optimizations to reduce software latency (e.g. bypassing the Linux IO scheduler and removing unnecessary context switches) and maximize concurrency (e.g., by removing all locks in the driver). These changes reduce latency by 62% compared to the standard Linux IO

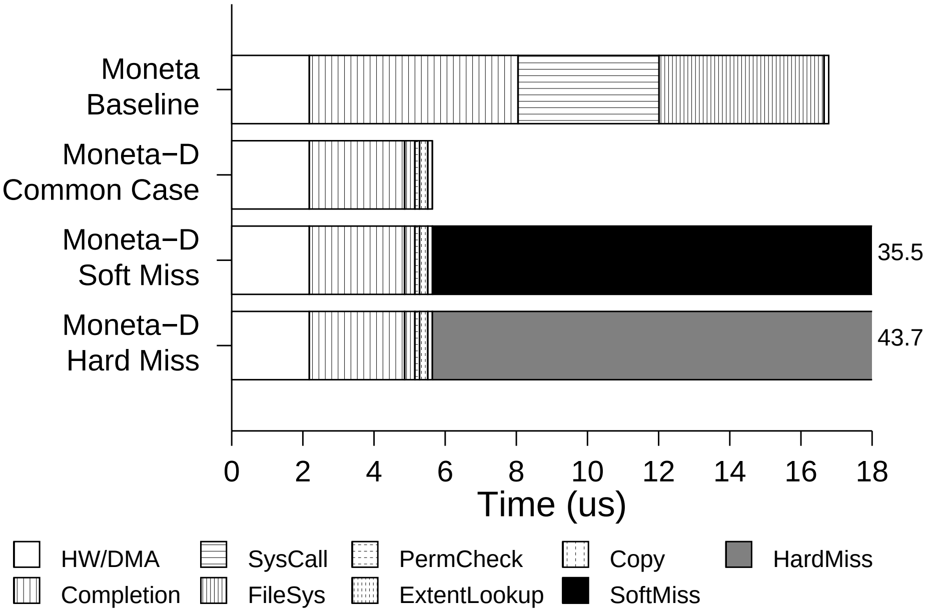

Figure 4.3. Software latencies required to manage permissions, tags, and user data all contribute to operation latency. DMA and copying values are for 512 byte accesses. The graph shows the latency breakdown for 1K extents and using DMA completion.

stack, but system call and file system overheads still account for 65% of the remaining software overheads.

The baseline design implements one channel that the operating system alone may access. It provides a set of configuration and command registers and targets a single DMA buffer in the kernel's address space.

Figure 4.3 shows the latency breakdown for 512 B reads and writes on Moneta-D. The hardware, DMA, and copy overheads are common across the baseline and the extensions we describe in this work. These, combined with the file system, system call, and interrupt processing overheads bring the total request latency in the baseline to 15.36 and 16.78 μs for reads and writes, respectively.

4.3.2 Virtual channels

Supporting virtual channels on Moneta-D requires replicating the control registers, tags, and DMA buffers mentioned above, while maintaining file coherency across multiple processes. This section describes the hardware and software implementation of virtual channels on Moneta-D. The dashed box in Figure 4.2 contains the components that implement virtual channels.

Control registers and data structures The interface for a channel comprises several memory-mapped hardware control registers and a shared memory segment. Together, these allow the kernel and the user space library to configure the channel, perform operations on it, and receive notifications when they complete.

Table 4.1 describes the most important control registers and the shared memory segment. The kernel's global registers allow the kernel to manage Moneta-D's functions that apply to multiple channels, such as error reporting and channel status. The user per-channel registers allow the process to access the hardware. Finally, the per-channel mapped memory shared by the kernel and user contains the channel's DMA buffer and data structures used to notify threads when operations complete. We discuss the role of these components in detail below.

In the non-virtualized system, a single set of control registers exposes Moneta's interface to the kernel. In the virtualized system, Moneta-D exposes 1024 channels, each with a private set of control registers located at a unique physical address. Reading or writing to any of these registers will send a PIO request to Moneta-D. Moneta-D uses the address bits to determine which channel the command targets. To give a process access to a particular channel, the kernel maps the registers for the channel into the process's address space. The unique mapping of physical addresses to channels allows Moneta-D to reliably know which process issued a particular request and prevents processes from accessing channels other than their own.

Request tags The baseline design supports 64 concurrent, outstanding requests. To maximize performance and concurrency, each channel needs its own set of tags. One option is to support 65,536 tags (64 tags for each of the 1024 channels) in hardware and statically partition them across the channels. In a custom ASIC implementation this might be possible, but in our FPGAs maintaining a request scoreboard of that size is not feasible at our 250 MHz clock frequency.

Instead, we provide each channel with 64 virtual tags and dynamically map them onto a set of 64 physical tags. The virtual tag number comprises the channel ID and the tag number encoded in the command word. The "retag" unit shown in Figure 4.2 assigns physical tags to requests by drawing physical tags from a hardware free tag list. If a physical tag is not available, the retag unit stalls until a request completes and releases its physical tag.

DMA buffer Each channel has a private 1 MB DMA buffer pinned in system DRAM that Moneta-D uses as the source and destination for writes and reads. The target DMA address for a request depends on its tag with each tag corresponding to one 16 KB slice of the channel's DMA buffer. If the process issues a command on tag *k*, the DMA transfer will start at the *k*th slice. The access that uses the tag can be larger than 16 KB, but it is the software's responsibility to not issue requests that overlap in the buffer. Moneta-D supports arbitrarily large DMA buffers, but since they must be contiguous in physical memory allocating bigger DMA buffers is challenging. Better kernel memory management would eliminate this problem.

Asynchronous interface Moneta-D's user space library provides asynchronous versions of its `pread()` and `pwrite()` calls (i.e., read/write to a given offset in a file). The asynchronous software interface allows applications to take advantage of the inherently asynchronous hardware by overlapping storage accesses with computation. For example, double buffering allows a single thread to load a block of data at the same time as it processes a different block. The asynchronous calls return immediately after issuing the request to the hardware and return an asynchronous IO state structure that identifies and tracks the request. The application can continue executing and only check for completion when it needs the data from a read or to know that a write has completed.

4.3.3 Translation and protection

The Moneta-D hardware, the user space library, and the operating system all work together to translate file-level accesses into hardware requests and to enforce permissions on those accesses. Translations between file offsets and physical storage locations occur in the user space library while the hardware performs permission checks. Below, we describe the role of both components and how they interact with the operating system and file system.

Hardware permission checks Moneta-D checks permissions on each request it receives after it translates virtual tags into physical tags ("Perm Check" in Figure 4.2). Since the check is on the critical path for every access, the checks can potentially limit Moneta-D's throughput. To maintain the baseline's throughput of 1.8 M IOPS, permissions checks must take no more than 500 ns.

Moneta-D must also cache a large amount of permission information in order to minimize the number of "misses" that will occur when the table overflows and the system must evict some entries. These *hard permission misses* require intervention from both the user space driver and the operating system to remedy, a process that can take 10s of microseconds (Figure 4.3).

To minimize the number of permission entries it must store for a given set of files, Moneta-D keeps extent-based permission information for each channel and merges entries for adjacent extents. All the channels share a single permission table with 16K entries. To avoid the need to scan the array linearly and to allow channels to dynamically share the table, Moneta-D arranges the extent information for each channel as a balanced red-black binary tree, with each node referring to a range of physical blocks and the permission bits for that range. The ”Root Table” (Figure 4.2) holds the location of the root of each channel’s tree. Using balanced trees keeps search times fast despite the potentially large size of the permission tree: The worst-case tree traversal time is 180 ns, and in practice the average latency is just 96 ns. With a linear scan, the worst-case time would exceed 65 μs.

To reduce hardware complexity, the operating system maintains the binary trees, and the hardware only performs look ups. The OS keeps a copy of the trees in system DRAM. When it needs to update Moneta-D’s permission table, it performs the updates on its copy and records the changes it made in a buffer. Moneta-D then reads the buffer via DMA, and applies the updates to the tree while temporarily suspending protection checking.

User space translation When the user space library receives a read or write request for a file on Moneta-D, it is responsible for translating the access address into a physical location in Moneta-D and issuing requests to the hardware.

The library maintains a translation map for each file descriptor it has open. The map has one entry per file extent. To perform a translation, the library looks up the target file location in the map. If the request spans multiple extents, the library will generate

Table 4.2. Software latencies required to manage permissions, tags, and user data all contribute to operation latency. DMA and copying values are for 512 byte accesses. Cells with a single value have the same latency for both read and write accesses.

Component		Latency R/W (μs)	
		1 extent	1K extents
Hardware + DMA		1.26 / 2.18	
Copy		0.17 / 0.13	
SW Extent lookup		0.12	0.23
HW Permission check		0.06	0.13
Soft miss handling		7.28	29.9
Hard miss handling		14.7	38.1
Permission update		3.23	3.26
File System	Baseline	4.21/4.64	
	Moneta-D	0.21/0.29	
System call	Baseline	3.90/3.96	
	Moneta-D	0.00/0.00	
Completion	Baseline (interrupts)	5.82 / 5.87	
	OS forwarding	2.71 / 2.36	
	DMA	2.32 / 2.68	
	issue-sleep	14.65 / 14.29	

multiple IO requests.

The library populates the map on-demand. If a look up fails to find an extent for a file offset, we say that a *soft permissions miss* has occurred. To service a soft miss, the library requests information for the extent containing the requested data from the operating system. The request returns the mapping information and propagates the extent's protection and physical location information to hardware.

Once translation is complete, the library issues the request to Moneta-D and waits for it complete. If the request succeeds, the operation is complete. Permission record eviction or an illegal request may cause the request to fail. In the case of an eviction, the permission entry is missing in hardware so the library reloads the permission record and tries again.

Permission management overheads Permission management and checking add some overhead to accesses to Moneta-D, but they require less time than the conventional system call and file system overheads that provide the same functions in conventional systems. Figure 4.3 shows the latencies for each component of an operation in the Moneta-D hardware and software. To measure them, we use a microbenchmark that performs 512 B random reads and writes to a channel with one permission record and another with 1000 records present. The microbenchmark selectively enables and disables different system components to measure their contribution to latency.

In the common case, accesses to Moneta-D incur software overhead in the user space library for the file offset-to-extent lookup. This requires between 0.12 and 0.23 μs, depending on the number of extents. The hardware permission check time is much faster – between 60 ns and 130 ns.

The miss costs are significantly higher: Handling a soft miss requires between 4.1 μs and 26.8 μs to retrieve extent information from the file system and 3.2 μs to update the permission tree in hardware. In total, a soft miss increases latency for a 512 B access by between 7.3 μs and 30 μs, depending on the number of extents in use. The hard miss adds another 7.7 μs of latency on average, because the user space library does not detect it until the initial request fails and reports an error.

In the best case, only one soft miss should occur per file extent. Whether hard misses are a problem depends on the number of processes actively using Moneta-D and the number of extents they are accessing. Since fragmented files will place more pressure on the permission table, the file system's approach to preventing fragmentation is important.

XFS uses aggressive optimizations to minimize the number of extents per file, but fragmentation is still a problem. We measured fragmentation on a 767 GB XFS file system that holds a heavily-used Postgres database [59] and found that, on average, each file contained 21 extents, and ninety-seven percent of files contained a single extent. However, several files on the file system contain 1000s of extents, and one database table contained 23,396.

We have implemented two strategies to deal with fragmentation. The first is to allocate space in sparse files in 1 MB chunks. When the library detects a write to an unallocated section of a file, it allocates space by writing up to 1 MB of zeroed data to that location before performing the user's request. This helps reduce fragmentation for workloads that perform small writes in sparse files. The second is to merge contiguous extents in the hardware permission table even if the file contents they contain are logically discontiguous in the file. This helps in the surprising number of cases in which XFS allocates discontinuous portions of a file in adjacent physical locations.

Figure 4.4 shows the benefits of merging permission entries. It shows aggregate throughput for a single process performing random 4 KB accesses to between 2048 and 32,768 extents. The two lines depict the workload running on Moneta-D with (labeled

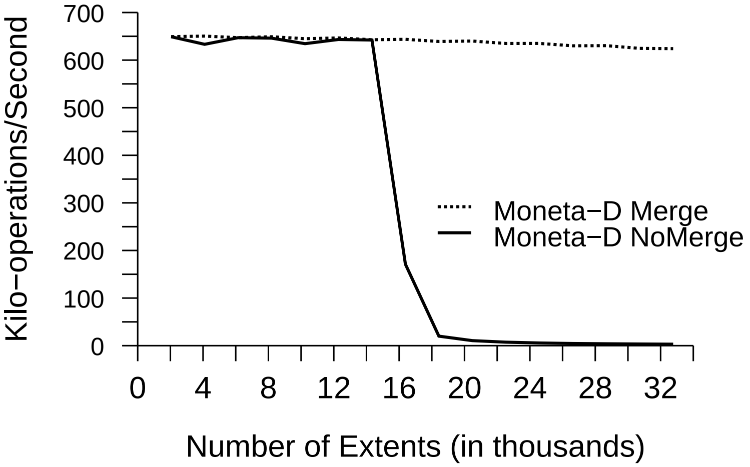

Figure 4.4. Merging permission entries improves performance because it allows Moneta-D to take advantage of logically discontinuous file extents that the file system allocates in physically adjacent storage.

”Moneta-D Merge”) and without (”Moneta-D NoMerge”) combining permission table entries. Moneta-D Merge combines entries if they belong to the same channel, represent data from the same file, have the same permission bits set, and cover physically adjacent blocks. Moneta-D NoMerge does not merge extents.

Throughput remains high for Moneta-D NoMerge when there are enough permission table entries to hold all the extent information. Once all 16K permission table entries are in use, throughput drops precipitously as the hard miss rate rises. For Moneta-D Merge, performance remains high even when the number of extents exceeds the permission table size by 2×, because many extents merge into a smaller number of entries.

Avoiding hard misses requires having a sufficient number of permission table entries available for the process accessing Moneta-D directly. There are (at least) three ways to achieve this. The first is to increase the permission table size. In a custom ASIC implementation this would not be difficult, although it would increase cost. The second is to detect over-subscription of the permission table and force some processes to use the conventional system call interface by evicting all their permission table entries, refusing to install new ones, and returning an error code informing the process of the change in policy. Finally, enhancements can be made to more aggressively avoid fragmention in the file system block allocator.

4.3.4 Completing requests and reporting errors

Modern hardware provides no mechanism for delivering an interrupt directly to a process, so virtualizing this aspect of the interface efficiently is difficult. Moneta-D supports three mechanisms for notifying a process when a command completes that tradeoff CPU efficiency and performance. Moneta-D also provides a scalable mechanism for reporting errors (e.g., permission check failures).

Forwarding interrupts The first notification scheme uses a traditional kernel interrupt handler to notify channels of request status through a shared memory page. Moneta-D's kernel driver receives the interrupt and reads the ChannelStatus register to determine which channels have completed requests. The kernel increments the CompletionCount variable for each of those channels.

After issuing a request, the user space library spins on both CompletionCount and the TagStatusTable entry for the request. Once the kernel increments CompletionCount one thread in the user space library sees the change and reads the per-channel TagStatusRegister from Moneta-D. This read also clears the register, ensuring that only one reader will see each bit that is set. The thread updates the TagStatusTable entries for all of the completed tags, signalling any other threads that are waiting for requests on the channel.

DMA completion The second command completion mechanism bypasses the operating system entirely. Rather than raise an interrupt, Moneta-D uses DMA to write the request's result code (i.e., success or an error) directly to the tag's entry in the channel's TagStatusTable. In this case, the thread spins only on the TagStatusTable.

Issue-sleep The above techniques require the issuing thread to spin. For large requests this is unwise, since the reduction in latency that spinning provides is small compared to overall request latency, and the spinning thread occupies a CPU, preventing it from doing useful work.

To avoid spinning, issue-sleep issues a request to hardware and then asks the OS to put it to sleep until the command completes. When an interrupt arrives, the OS restarts the thread and returns the result code for the operation. This approach incurs the system call overhead but avoids the file system overhead, since permission checks still occur in hardware. The system call also occurs *in parallel* with the access.

It is possible to combine issue-sleep on the same channel with DMA completions, since the latter does not require interrupts. This allows the user library to trade-off between completion speed and CPU utilization. A bit in the command word that initiates a requests tells Moneta-D which completion technique to use. We explore this trade-off below.

Reporting errors Moving permission checks into hardware and virtualizing Moneta's interface complicates the process of reporting errors. Moneta-D uses different mechanisms to report errors depending on which completion technique the request is using.

For interrupt forwarding and issue-sleep, the hardware enqueues the type of error along with its virtual tag number and channel ID in a hardware error queue. It then sets the error bit in the ChannelStatus register and raises an interrupt.

The kernel detects the error when it reads the ChannelStatus register. If the

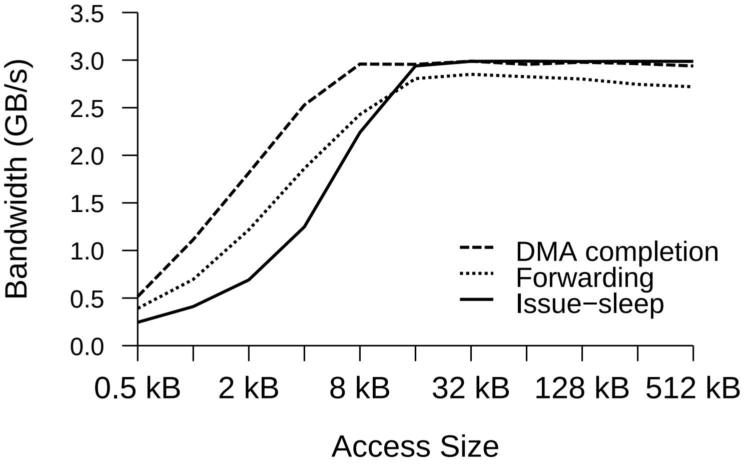

Figure 4.5. This graph compares performance for our three completion strategies for 32 threads performing a 50/50 mix of random reads and writes. It measures maximum sustained bandwidth. For small accesses, DMA completion is the best alternative by both measures. For larger accesses, however, issue-sleep enjoys a large advantage in efficiency.

error bit is set, it extracts the error details from the queue by reading repeatedly from the ERRORQUEUE register. Each read dequeues an entry from the error queue. For interrupt forwarding the kernel copies the error codes into the TAGSTATUSREGISTERs for the affected channels. For issue-sleep completion it returns the error when it wakes up the sleeping thread. The kernel reads from ERRORQUEUE until it returns zero.

For DMA completion, the hardware writes the result code for the operation directly into the TAGSTATUSTABLE when the operation completes.

Completion technique performance The four completion method lines of Figure 4.3 measure the latency of each complation strategy in addition to the interrupt processing overhead for the baseline Moneta design. Interrupt forwarding and DMA completion all have similar latencies – between 2.5 and 2.7 μs. Issue-sleep takes over 14 μs, but for large requests, where issue-sleep is most useful, latency is less important.

Figures 4.5 and 4.6 compare the performance of the three completion techniques.

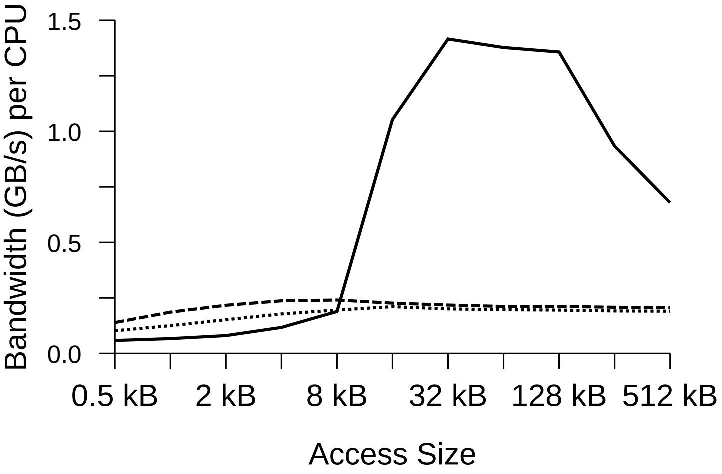

Figure 4.6. This graph compares CPU behavior for our three completion strategies for 32 threads performing a 50/50 mix of random reads and writes. It measures efficiency as the ratio of bandwidth to CPU utilization. For small accesses, DMA completion is the best alternative by both measures. For larger accesses, however, issue-sleep enjoys a large advantage in efficiency.

The data are for 32 threads performing random accesses of the size given on the horizontal axis. Half of the accesses are reads and half are writes. Figure 4.5 measures aggregate throughput and shows that DMA completion outperforms the other schemes by between 21% and 171%, for accesses up to 8 KB. Issue-sleep performs poorly for small accesses, but for larger accesses its performance is similar to interrupt forwarding. We use DMA completion throughout the rest of this work.

Figure 4.6 measures efficiency in terms of GB/s of bandwidth per CPU. The two spinning-based techniques fare poorly for large requests. Issue-sleep does much better and can deliver up to 7× more bandwidth per CPU. The drop in issue-sleep performance for requests over 128 KB is an artifact of contention for tags in our microbenchmark: Threads spin while waiting for a tag to become available and yield the processor between each check. Since our microbenchmark does not do any useful work, the kernel immediately reschedules the same thread. In a real application, another thread would likely run instead, reducing the impact of the spinning thread.

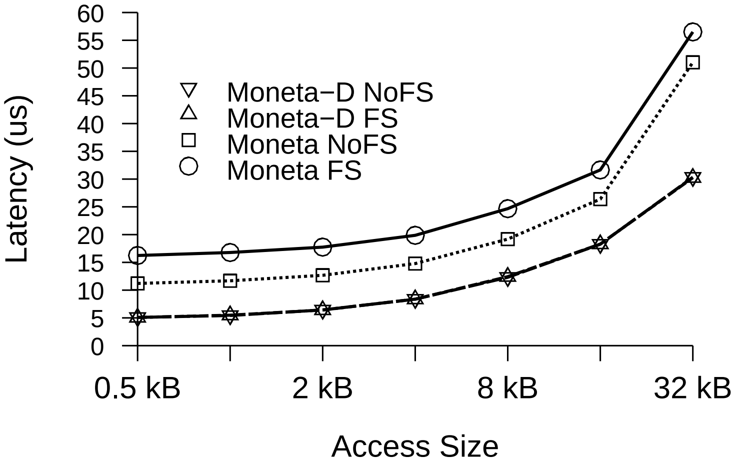

Figure 4.7. Moneta-D's user space interface eliminates most of the file system and operating system overheads to reduce file system access latency by between 42% and 69%.

The data show that DMA completion is a good choice for all request sizes, but that issue-sleep has a slight advantage for CPU-intensive workloads. Issue-sleep is the best choice for accesses of 8 KB or larger because it is more CPU-efficient.

4.4 Results

Moneta-D's virtualized interface reduces both file and operating system overhead, but it also introduces new sources of potential latency as described in Section 4.3. This section quantifies the overall impact of these changes on Moneta-D's performance using an IO microbenchmark and several database applications. We also evaluate the benefits of using asynchronous IO on Moneta-D rather than the conventional synchronous operations.

4.4.1 Operation latency

Figure 4.7 shows how end-to-end single thread access latency varies over a range of write request sizes from 512 B to 32 KB on the baseline Moneta design and Moneta-

D. Read latencies are similar. The graph shows data for accesses running with 1000 permission table entries installed. We collect these data using XDD [109], a flexible IO benchmarking tool. Moneta-D extends the baseline Moneta's performance by a wide margin, while Moneta outperforms state-of-the-art flash-based SSDs by up to 8.7×, with a harmonic mean speedup of 2.1× on a range of file system, paging, and database workloads [12].

Figure 4.7 shows that Moneta-D eliminates most file system and operating system overheads from requests of all sizes, since the lines for Moenta-D FS and NoFS lay on top of each other. Figure 4.3 provides details on where the latency savings come from for small requests. Assuming the access hits in the permission table, Moneta-D all of this, reducing latency by 60%. Reducing software overheads for small (512 B) requests is especially beneficial because as request size decreases, hardware latency decreases and software latency remains constant.

4.4.2 Raw bandwidth

Since removing the operating and file systems from common case accesses reduces software overhead per IO operation, it also increases throughput, especially for small accesses. Figures 4.8 and 4.9 compares the bandwidth for Moneta-D and baseline Moneta with and without the file system. For writes, the impact of virtualization is large: Adding a file system reduces performance for the original Moneta system by up to 13×, but adding a file system to Moneta-D has almost no effect. Moneta-D eliminates the gap for reads as well, although the impact of the file system on the baseline is smaller (at most 34%).

Reducing software overheads also increases the number of IO operations the system can complete per second, because the system must do less work for each operation. For small write accesses with a file system, throughput improves by 26×, and Moneta-D

Table 4.3. We use eight database benchmarks and workloads to evaluate Moneta-D. These benchmarks include both lightweight Berkeley-DB database transactions and simple and complex transactions against heavyweight SQL databases.

Name	Data footprint	Description
Berkeley-DB Btree	45 GB	Transactional updates to a B+tree key/value store
Berkeley-DB Hash	41 GB	Transactional updates to a hash table key/value store
MySQL-Simple	46 GB	Single value random select queries on MySQL database
MySQL-Update	46 GB	Single value random update queries on MySQL database
MySQL-Complex	46 GB	Mix of read/write queries in transactions on MySQL database
PGSQL-Simple	55 GB	Single value random select queries on Postgres database
PGSQL-Update	55 GB	Single value random update queries on Postgres database
PGSQL-Complex	55 GB	Mix of read/write queries in transactions on Postgres database

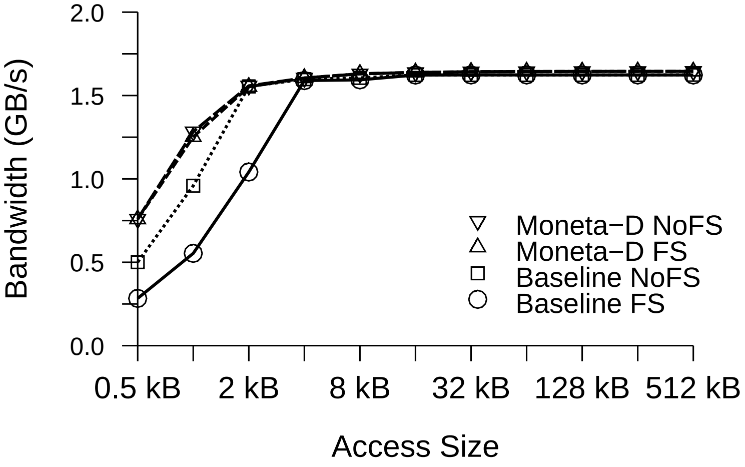

Figure 4.8. Each pair of lines compares bandwidth with and without the file system for writes for the baseline system and Moneta-D. The data show that giving the applications direct access to the hardware nearly eliminates the performance penalty of using a file system and the cost of entering the operating system.

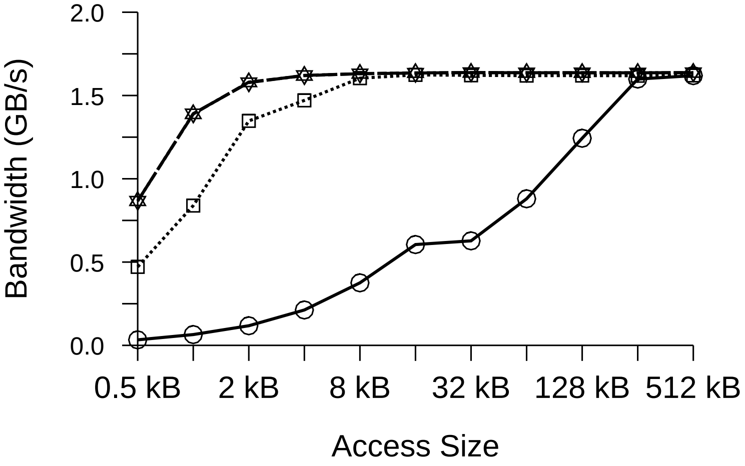

Figure 4.9. Each pair of lines compares bandwidth with and without the file system for writes for the baseline system and Moneta-D. The data show that giving the applications direct access to the hardware nearly eliminates the performance penalty of using a file system and the cost of entering the operating system.

sustains 1.8 M 512 B IO operations per second.

4.4.3 Application level performance

Table 4.3 describes the workloads we use to evaluate the application level performance of Moneta-D compared to the baseline Moneta design. The first two workloads are simple database applications that perform random single-value updates to a large key-value store in Berkeley-DB, backed either by a B+tree or a hash table. The six MySQL and PGSQL workloads consist of full OLTP database servers that aggressively optimize storage accesses and have strong consistency requirements. They are running the OLTP portion of the SysBench benchmark suite [96] which includes a variety of OLTP operations including read-only lookups, single-value increments, and complex transactions with multiple lookups and updates.

Table 4.4 shows the performance results for baseline Moneta and the Moneta-D systems for all of our workloads. We also include performance numbers from a FusionIO 80 GB Flash SSD for comparison. Moneta-D speeds up the Berkeley-DB applications by between 2.6× and 5.7× in terms of operations/second, compared to baseline Moneta and by between 5.3× and 9.8× compared to FusionIO. We attribute the difference in performance between these two workloads to higher data structure contention in the B+tree database implementation.

Figure 4.10 shows application performance speedup for varying thread counts from 1 to 16 for the Berkeley-DB and complex MySQL and Postgres databases. Other MySQL and Postgres results are similar. The results are normalized to 1-thread baseline Moneta performance. Baseline Moneta out-performs FusionIO for all thread counts across all of workloads, while Moneta-D provides additional speedup. Increasing thread counts on Moneta-D provides significantly more speedup than on Moneta or FusionIO for the Berkeley-DB workloads, and improves scaling on PGSQL-Complex.

Table 4.4. Moneta-D provides significant speedups compared to baseline Moneta and FusionIO across a range of workloads. The Berkeley-DB workloads benefit more directly from the increased IO throughput, while the full SQL databases see large gains for write intensive queries. All of the data use the best performing thread count (between 1 and 16) for each workload. FusionIO performance is provided for reference.

	Raw Performance			Speedup of Moneta-D vs.	
Workload	FusionIO	Moneta	Moneta-D	FusionIO	Moneta
Berkeley-DB Btree	4066 ops/s	8202 ops/s	21652 ops/s	5.3 ×	2.6 ×
Berkeley-DB Hash	6349 ops/s	10988 ops/s	62124 ops/s	9.8 ×	5.7 ×
MySQL-Simple	13155 ops/s	13840 ops/s	15498 ops/s	1.2 ×	1.1 ×
MySQL-Update	1521 ops/s	1810 ops/s	2613 ops/s	1.7 ×	1.4 ×
MySQL-Complex	390 ops/s	586 ops/s	866 ops/s	2.2 ×	1.5 ×
PGSQL-Simple	23697 ops/s	49854 ops/s	63308 ops/s	2.7 ×	1.3 ×
PGSQL-Update	2132 ops/s	2523 ops/s	5073 ops/s	2.4 ×	2.0 ×
PGSQL-Complex	569 ops/s	1190 ops/s	1809 ops/s	3.2 ×	1.5 ×
			Harmonic mean	2.4 ×	1.7 ×

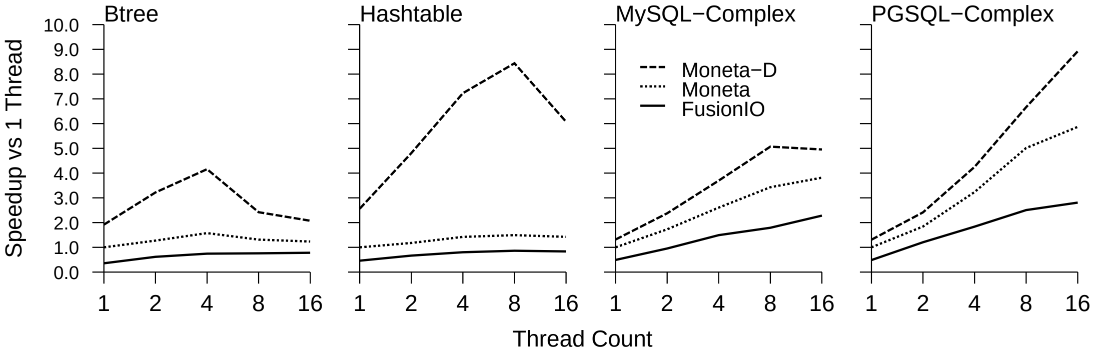

Figure 4.10. For most workloads, Moneta-D allows application performance to scale better with additional threads, especially for workloads with little inter-thread contention, like Hashtable. Changing Hashtable from 1 to 8 threads on Moneta-D increases performance by 2.2 ×more than the same change on either FusionIO or baseline Moneta.

The larger database applications, MySQL and Postgres, see performance improvements from 1.1× to 2.0× under Moneta-D, compared to baseline Moneta. The data show that for these workloads, write-intensive operations benefit most from Moneta-D, with transaction throughput increases of between 1.4× to 2.0×. Read-only queries also see benefits but the gains are smaller — only 1.1× to 1.3×. This is consistent with Moneta-D's smaller improvements for read request throughput.

We found that Postgres produces access patterns that interact poorly with Moneta-D, and that application level optimizations enable better performance. Postgres includes many small extensions to the files that contain its database tables. With Moneta-D these file extensions each result in a soft miss. Since Postgres extends the file on almost all write accesses, these soft misses eliminate Moneta-D's performance gains. Pre-allocating zeroed out data files before starting the database server enables Postgres to take full advantage of Moneta-D. Although Moneta-D requires no application level changes to function, this result suggests that, large performance improvements could result from additional optimizations at the application level, such as allocating large blocks in the file system rather than many small file extensions.

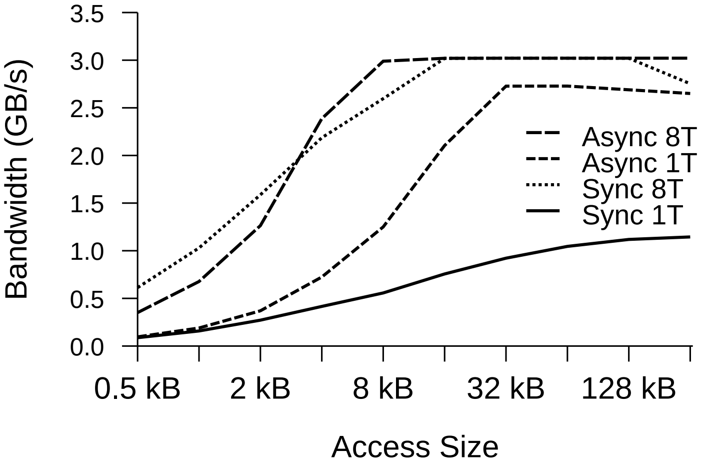

Figure 4.11. Moneta-D's asynchronous interface improves single threaded performance by up to 3.5× by eliminating time spent waiting for completions.

4.4.4 Asynchronous IO

Providing an asynchronous IO interface to Moneta-D allows applications to take advantage of its inherently asynchronous hardware interface. Figures 4.11 and 4.12 compares the performance of Moneta-D with and without asynchronous IO. Figure 4.11 shows sustained bandwidth for the synchronous and asynchronous interfaces with 1 and 8 threads. Asynchronous operations increase throughput by between 1.1× and 3.0× on access sizes of 512 B to 256 KB when using 1 thread. With 8 threads, asynchronous operations boost performance for requests of 4 KB or larger. Small request performance suffers from software overheads resulting from maintaining asynchronous request data structures and increased contention during tag allocation.

Figure 4.12 shows the efficiency gains from using asynchronous requests on 16 KB accesses for varying numbers of threads. The data show that for one thread, asynchronous requests are 2.8× more efficient than synchronous requests with respect to the amount of bandwidth per CPU. As the number of threads increases, the asynchronous accesses slowly lose their efficiency advantage compared to synchronous accesses. As the

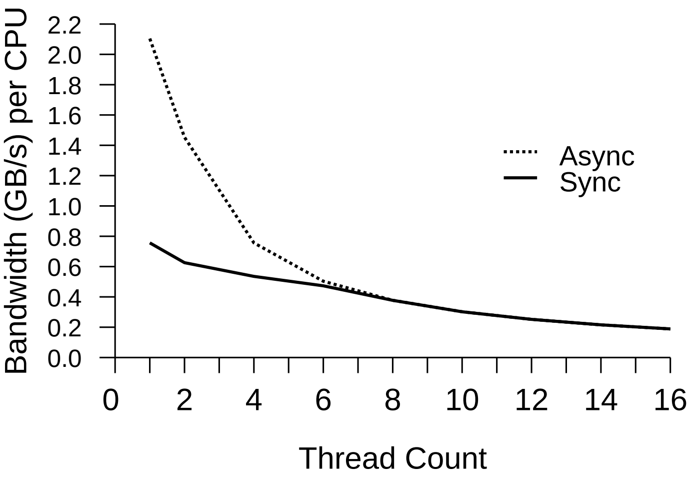

Figure 4.12. measures efficiency in terms of bandwidth per CPU with synchronous and asynchronous 16 KB accesses for varying numbers of threads.

number of threads increases, the per-thread performance decreases because of increased contention for hardware bandwidth and tags.

To understand the application-level impact of an asynchronous interface, we modified the ADPCM codec from Mediabench [54] to use Moneta-D's asynchronous IO interface and then used it to decode a 100 MB file. Using the asynchronous IO interface results in an 1.4× speedup over the basic Moneta-D interface. By using three buffers, ADPCM can process one block while reading in another and writing out a third. ADPCM's performance demonstrates how overlapping data accesses with data processing enables significant gains. In this case, Moneta-D transformed an IO bound workload into a CPU bound one, shifting from 41% CPU utilization for one thread on the baseline Moneta system to 99% CPU utilization with the asynchronous interface.

4.5 Summary

As emerging non-volatile memory technologies shrink storage hardware latencies, hardware interfaces and system software must adapt or risk squandering the performance these memories offer. Moneta-D avoides this danger by moving file system permission checks into hardware and using an untrusted, user space driver to issue requests. These changes reduce latency for 4 KB write requests through the file system by up to 58% and increase throughput for the same requests by 7.6×. Reads are 60% faster. These increases in raw performance translate into large application level gains. Throughput for an OLTP database workload increased 2.0× and our Berkeley-DB based workloads sped up by 5.7×. Asynchronous IO support provides 5.5× better 4 KB access throughput with 1 thread, and 2.8× better efficiency for 512-B operations, resulting in a 1.7× throughput improvement for a streaming application. Overall, out results demonstrate the importance of eliminating software overheads in IO-intensive applications that will use these emerging memories and point to several opportunities to improve performance further by modifying the applications themselves.

Acknowledgements

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 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. The dissertation author was the primary 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.

Chapter 5

Distributed Storage

Modern storage systems rely on complex software and interconnects to provide scalable, reliable access to large amounts of data across multiple machines. In conventional, disk-based storage systems the overheads from file systems, remote block device protocols (e.g., iSCSI and Fibre Channel), and network stacks are tiny compared to the storage media access time, so software overheads do not limit scalability or performance.

The emergence of non-volatile, solid-state memories (e.g., NAND flash, phase change memory, and spin-torque MRAM) changes this landscape completely by dramatically improving storage media performance. As a result, software shifts from being the least important component in overall storage system performance to being a critical bottleneck.

The software costs in scalable storage systems arise because, until recently, designers could freely compose existing system components to implement new and useful storage system features. For instance, the software stack running on a typical, commodity storage area network (SAN) will include a file system (e.g., GFS [94]), a logical volume manager (e.g., LVM), remote block transport layer client (e.g., the iSCSI initiator), the client side TCP/IP network stack, the server network stack, the remote block device server (e.g., the iSCSI target), and the block device driver. The combination of these layers (in addition to the other operating system overheads) adds 288 μs of latency to a 4 kB read. Hardware accelerated solutions (e.g., Fibre Channel) eliminate some of these costs, but still add over 90 μs per access.

Software latencies of hundreds of microseconds are inconsequential for disk accesses (with a hardware latency measured in many milliseconds), but storage systems like Moneta and Moneta-Direct, introduced in the previous chapters, will deliver $<$10 μs latencies. At those speeds, conventional software and block transport latencies will cripple performance.

This chapter describes a new SAN architecture called *QuickSAN*. QuickSAN re-examines the hardware and software structure of distributed storage systems to minimize software overheads and let the performance of the underlying storage technology shine through.

QuickSAN improves SAN performance in two ways. First, it provides a very low-latency, hardware-accelerated block transport mechanism (based on 10 Gbit ethernet) and integrates it directly into an SSD. Second, it extends the OS bypass mechanism introduced in the previous chapter to allow an application to access remote data without (in most cases) any intervention from the operating system while still enforcing file system protections.

We examine two different structures for QuickSAN. *Centralized QuickSAN* systems mirror conventional SAN topologies that use a centralized data storage server. *Distributed QuickSAN* enables distributed storage by placing one slice of a globally shared storage space in each client node. As a result, each node has very fast access to its local slice, and retains fast, direct access to the slices at the other nodes as well. In both architectures, clients see storage as a single, shared storage device and access data via a shared-disk file system and the network.

We evaluate QuickSAN using GFS2 [32] and show that it reduces software and block transfer overheads by 94%, reduces overall latency by 93% and improves

Table 5.1. Software stack and block transport latencies add overhead to storage accesses. Software-based SAN interfaces like iSCSI takes 100s of microseconds. Fibre Channel reduces this costs, but the additional latency is still high compared to the raw performance of fast non-volatile memories. The Fibre Channel numbers are estimates based on [3] and measurements of QuickSAN.

	Existing interfaces		QuickSAN	
Component	Latency R/W (μs)		Latency R/W (μs)	
	iSCSI	Fibre Channel	Kernel	Direct
OS Entry/Exit & Block IO	4.0 / 3.4		3.8 / 3.4	0.3 / 0.3
GFS2	0.1 / 1.5		1.4 / 2.3	n/a
Block transport	284.0 / 207.7	86.0 / 85.9	17.0/16.9	17.0/16.9
4 kB non-media total	288.1 / 212.6	90.1 / 90.8	22.2/22.6	17.2/17.1

bandwidth by 14.7× for 4 KB requests. We also show that leveraging the fast access to local storage that distributed QuickSAN systems provide can improve performance by up to 3.0× for distributed sorting, the key bottleneck in MapReduce systems. We also measure QuickSAN's scalability, its impact on energy efficiency, and the impact of adding mirroring to improve reliability. Finally, we show how distributed QuickSAN can improve cluster-level efficiency under varying load.

The remainder of this chapter is organized as follows. Section 5.1 describes existing SAN-based storage architectures and quantifies the software latencies they incur. Section 5.2 describes QuickSAN's architecture and implementation in detail. Section 5.3 places QuickSAN in the context of previous efforts. Section 5.4 evaluates both centralized and distributed QuickSAN to compare their performance on a range of storage workloads. Section 5.5 presents our conclusions.

5.1 Motivation

Non-volatile memories that offer order of magnitude increases in performance require us to reshape the architecture of storage area networks (SANs). SAN-based storage systems suffer from two sources of overhead that can obscure the performance of the underlying storage. The first is the cost of the device-independent layers of the software stack (e.g., the file system), and the second is the *block transport cost*, the combination of software and interconnect latency required to access remote storage. These costs range from 10s to 100s of microseconds in modern systems.

As non-volatile memories become more prevalent, the cost of these two components grows relative to the cost of the underlying storage. For disks, with latencies typically between 7 and 15 ms, the costs of these layers is almost irrelevant. Flash-based SSDs, however, have access times under 100 μs and SSDs based on more advanced memories will have $<$10 μs latencies. In these systems, the cost of conventional storage software and block transport will completely dominate storage costs.

This section describes the sources of these overheads in more detail and places them in context of state-of-the-art solid-state storage devices.

5.1.1 Storage overheads

Figure 5.1 shows the architecture of two typical SAN-based shared-disk remote storage systems. Both run under Linux and GFS2 [32], a distributed, shared-disk file system. In Figure 5.1(a) a software implementation of the iSCSI protocol provides block transport. An iSCSI target (i.e., server) exports a block-based interface to a local storage system. Multiple iSCSI client initiators (i.e., clients) connect to the target, providing a local interface to the remote device.

Fibre Channel, Figure 5.1(b), provides block transport by replacing the software below the block interface with a specialized SAN card that communicates with Fibre Channel storage devices. Fibre Channel devices can be as simple as a hard drive or as complex as a multi-petabyte, multi-tenancy storage system, but in this work we focus on performance-optimized storage devices (e.g., the Flash Memory Arrays [101] from Violin Memory).

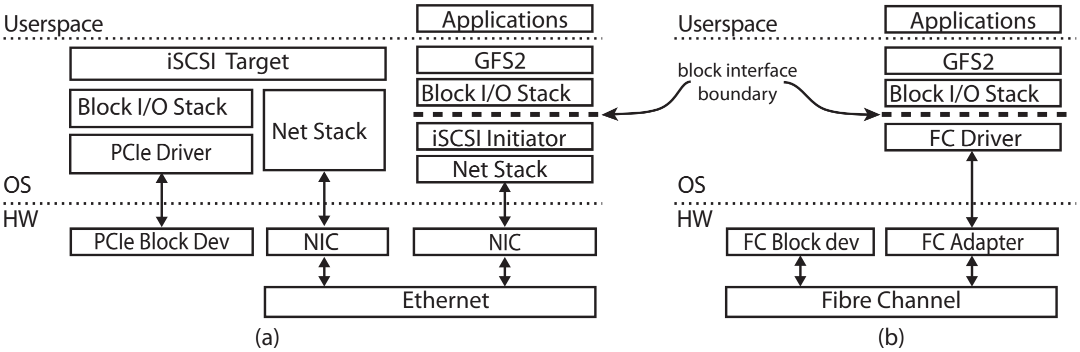

Figure 5.1. Existing SAN systems use software-based architectures like software iSCSI (a) or specialized hardware solutions like Fibre Channel (b). Both hide ample complexity behind the block device interface they present to the rest of the system. That complexity and the software that sits above it in OS both add significant overheads to storage access.

The values in Table 5.1 measure the OS and block transport latency for a 4 kB request from the applications perspective for iSCSI and Fibre Channel. The first three rows are the generic OS storage stack. The fourth row measures the minimum latency for a remote storage access starting at the block interface boundary in Figure 5.1, and excluding the cost of the actual storage device. These latencies seem small compared to the 7-11 ms required for the disk access, but next-generation SSDs fundamentally alter this balance.

5.1.2 The Impact of Fast SSDs

Fast non-volatile memories are already influencing the design of storage devices in the industry. As faster memories become available the cost of software and hardware overheads on storage accesses will become more pronounced.

Commercially available NAND flash-based SSDs offer access latencies of tens of microseconds, and research prototypes targeting more advanced memory technologies have demonstrated latencies as low as 4-5 μs [13].

Violin Memory's latest 6000 Series Flash Memory Arrays [101] can perform read accesses in 80-100 μs and writes in 20 μs with aggressive buffering. FusionIO's latest

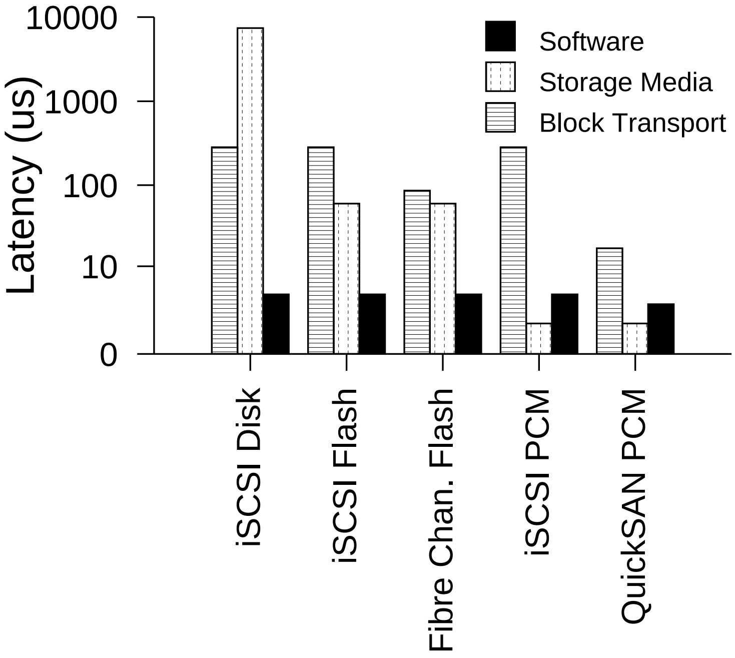

Figure 5.2. As storage shifts from disks to solid-state media, software and block transport account for a growing portion of overall latency. This shift requires us to rethink conventional SAN architectures.

ioDrives [29] do even better: 68 μs for reads and 15 μs for writes, according to their data sheets. Similar drives from Intel [44] report 65 μs for reads and writes.

At least two groups [12, 13, 112] have built prototype SSDs that use DRAM to emulate next-generation memories like phase-change memories, spin-torque MRAMs, and the memristor. These devices can achieve raw hardware latencies of between 4 and 5 μs [13, 112].

Figure 5.2 illustrates the shift in balance between the cost of underlying storage and the cost of software and the block transport layers. For a disk-based SAN, these overheads represent between 1.2 and 3.8% of total latency. For flash-based SSDs the percentage climbs as high as 59.6%, and for more advanced memories software overheads will account for 98.6-99.5% of latency.

Existing software stacks and block transport layers do not provide the low-latency access that fast SSDs require to realize their full potential. The next section describes our design for QuickSAN that removes most of those costs to expose the full performance of SSDs to software.

5.2 QuickSAN

This section describes the hardware and software components of QuickSAN. The QuickSAN hardware adds a high-speed network interface to a PCIe-attached SSD, so SSDs can communicate directly with one another and retrieve remote data to satisfy local IO requests. The QuickSAN software components leverage this capability (along with existing shared-disk file systems) to eliminate most of the software costs described in the previous section.

5.2.1 QuickSAN Overview

We explore two different architectures for QuickSAN storage systems. The first resembles a conventional SAN storage system based on Fibre Channel or iSCSI. A central server (at left in Figure 5.3) exports a shared block storage device, and client machines access that device via the network. We refer to this as the *centralized* architecture.

The second, *distributed*, architecture replaces the central server with a distributed set of SSDs, one at each client (Figure 5.3, right). The distributed SSDs combine to expose a single, shared block device that any of the clients can access via the network.

To explore ways of reducing software overheads in these two architectures, we have implemented two hardware devices: A customized network interface card (the QuickSAN NIC) and a custom SSD (the QuickSAN SSD) that integrates both solid state storage and QuickSAN NIC functionality. In some of our experiments we also disable the network interface on the QuickSAN SSD, turning it into a normal SSD.

The QuickSAN NIC is similar to a Fibre Channel card: It exports a block device

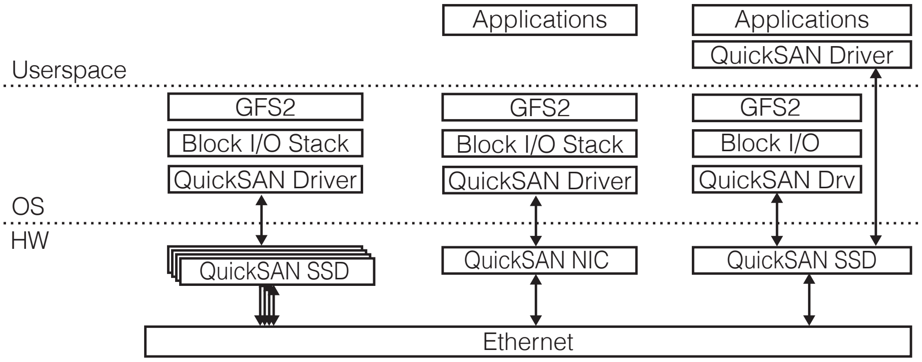

Figure 5.3. QuickSAN supports multiple storage topologies and two software interfaces. At left, a single machine hosts multiple QuickSAN SSDs, acting a central block server. The center machine hosts a QuickSAN NIC that provides access to remote SSDs. The machine at right hosts a single SSD and is poised to access its local (for maximum performance) or remote data via the userspace interface.

interface and forwards read and write requests over a network to a remote storage device (in our case, a QuickSAN SSD). The QuickSAN SSD responds to remote requests without communicating with the operating system, so it resembles a Fibre Channel client and target device.

The QuickSAN SSD also exports a block device interface to the host system. The interface's block address space includes the SSD's local storage and the storage on the other QuickSAN SSDs it is configured to communicate with. The QuickSAN SSD services accesses to local storage directly and forwards requests to external storage to the appropriate QuickSAN SSD.

This organization allows QuickSAN SSDs to communicate in a peer-to-peer fashion, with each SSD seeing the same global storage address space. This mode of operation is not possible with conventional SAN hardware.

QuickSAN further reduces software latency by applying the OS-bypass techniques described in [13]. These techniques allow applications to bypass the system call and file system overheads for common-case read and write operations while still enforcing file system protections.

The following subsections describe the implementation of the QuickSAN SSD and NIC in detail, and then review the OS bypass mechanism and its implications for file system semantics.

5.2.2 The QuickSAN SSD and NIC

The QuickSAN NIC and the QuickSAN SSD share many aspects of their design and functionality. Below, we describe the common elements of both designs and then the SSD- and NIC-specific hardware. Then we discuss the hardware platform we used to implement them both.

Common elements The left half of Figure 5.4 contains the common elements of the SSD and NIC designs. These implement a storage-like interface that the host system can access. It supports read and write operations from/to arbitrary locations and of arbitrary sizes (i.e., accesses do not need to be aligned or block-sized) in 64 bit address space.

The hardware includes the host-facing PIO and DMA interfaces, the request queues, the request scoreboard, and internal buffers. These components are responsible for accepting IO requests, executing them, and notifying the host when they are complete. Communication with the host occurs over a PCIe 1.1x8 interface, which runs at 2 GB/s, full-duplex. This section also includes the virtualization and permission enforcement hardware described below (and in detail in [13]).

Data storage The storage-related portions of the design are shown in the top-right of Figure 5.4. The SSD contains eight high-performance, low-latency non-volatile memory controllers attached to an internal ring network. The SSD's local storage address space is striped across these controllers with a stripe size of 8 kB.

In this work, the SSD uses emulated phase change memory, with the latencies from [53] — 48 ns and 150 ns for array reads and writes, respectively. The array uses

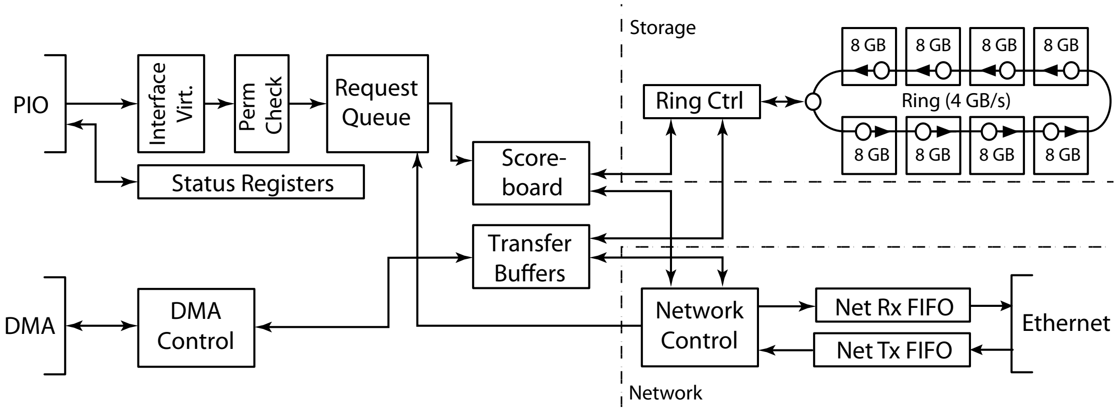

Figure 5.4. The QuickSAN NIC and SSD both expose a virtualized, storage interface to the host system via PCIe (at left). The network interface attaches to a 10 Gbit network port (bottom-right), while the QuickSAN SSD adds 64 GB of non-volatile storage (top-left). The device services requests that arrive via either the network or host interface and forwards requests for remote storage over the network.

start-gap wear leveling [77] to distribute wear across the phase change memory and maximize lifetime.

The network interface QuickSAN's network interface communicates over a standard 10 Gbit CX4 ethernet port, providing connectivity to other QuickSAN devices on the same network. The network interfaces plays two roles: It is a source of requests, like the PCIe link from the host, and it is also a target for data transfers, like the memory controllers.

The link allows QuickSAN to service remote requests without operating system interaction on the remote node and even allows access to storage on a remote node when that node's CPU is powered off (assuming the SSD has an independent power supply).

QuickSAN requires a lossless interconnect. To ensure reliable delivery of network packets, QuickSAN uses ethernet but employs flow control as specified in the IEEE 802.3x standard. Fibre Channel over ethernet (FCoE) uses the same standards to provide reliable delivery. Ethernet flow control can interact poorly with TCP/IP's own flow control and is usually disabled on data networks, but a more recent standard, 802.1qbb, extends flow control to cover multiple classes of traffic and alleviates these concerns. It is part of the “data center bridging” standard pushing for converged storage and data networks in data centers [18].

Ethernet flow control provides the necessary reliability guarantees that QuickSAN needs, but it runs the risk of introducing deadlock into the network if insufficient buffering is available. For example, deadlock can occur if an SSD must pause incoming traffic due to insufficient buffer space, but it also requires an incoming response from the network before it can clear enough buffer space to unpause the link.

We have carefully designed QuickSAN’s network interface to ensure that it always handles incoming traffic without needing to send an immediate response on the network, thus breaking the conditions necessary for deadlock. We guarantee sufficient buffer space for (or can handle without buffering) all incoming small (non-payload bearing) incoming messages. Read requests allocate buffers at the source before sending the request, and a dedicated buffer is available for incoming write requests, which QuickSAN clears without needing to send an outgoing packet.

Userspace access The host-facing interface of the QuickSAN SSD and NIC includes support for virtualization and permissions enforcement, similar to the mechanisms presented in [13]. These mechanisms allow applications to issue read and write requests directly to the hardware without operating system intervention while preserving file system permissions. This eliminates, for most operations, the overheads related to those software layers.

The virtualization support exposes multiple, virtual hardware interfaces to the NIC or SSD, and the operating system assigns one virtual interface to each process. This enables the process to issue and complete IO operations. A user space library interposes on file operations to provide an almost (see below) POSIX-compliant interface to the device.

The permissions mechanism associates a permissions table with each virtual interface. The hardware checks each request that arrives via the PCIe interface against the permission table for the appropriate interface. If the interface does not have permission to perform the access, the hardware notifies the process that the request has failed.

The operating system populates the permission table on behalf of the process by consulting the file system to retrieve file layout and permission information.

Permission checks in QuickSAN occur at the source NIC or SSD rather than at the destination. This represents a trade off between scalability and security. If permission checks happened at the destination, the destination SSD would need to store permission records for all remote requestors and would need to be aware of remote virtual interfaces. Checking permissions at the source allows the number of system wide permission entries and virtual interfaces to scale with the number of clients. A consequence is that QuickSAN SSDs trust external requests, an acceptable trade-off in most clustered environments.

Since the userspace interface provides direct access to storage, it can lead to violations of the atomicity guarantees that POSIX requires. For local storage devices hardware support for atomic writes [19] can restore atomicity. For distributed storage systems, however, achieving atomicity requires expensive distributed locking protocols, and implementing these protocols in hardware seems unwieldly.

Rather than provide atomicity in hardware, we make the observation that most applications that require strong consistency guarantees and actively share files (e.g., databases) rely on higher-level, application-specific locking primitives. Therefore, we relax the atomicity constraint for accesses through the userspace interface. If applications require the stronger guarantees, they can perform accesses via the file system and rely on it to provide atomicity guarantees. The userspace library automatically uses the file system for append operations since they must be atomic.

Implementation We implemented QuickSAN on the BEE3 prototyping platform [7]. The BEE3 provides four FPGAs which each host 16 GB of DDR2 DRAM, two 10 Gbit ethernet ports, and one PCIe interface. We use one of the ethernet ports and the PCIe link on one FPGA for external connectivity. The design runs at 250 MHz.

To emulate the performance of phase change memory using DRAM, we used a modified DRAM controller that allows us to set the read and write latency to the values given earlier.

5.2.3 QuickSAN software

Aside from the userspace libray described above, most of the software that QuickSAN requires is already commonly available on clustered systems.

QuickSAN's globally accessible storage address space can play host to conventional, shared-disk file systems. We have successfully run both GFS2 [32] and OCSF2 [71] on both the distributed and local QuickSAN configurations. We expect it would work with most other shared-disk file systems as well. Using the userspace interface requires that the kernel be able to query the file system for file layout information (e.g., via the fiemap ioctl), but this support is common across many file systems.

In a production system, a configuration utility would set control registers in each QuickSAN device to describe how the global storage address space maps across the QuickSAN devices. The hardware uses this table to route requests to remote devices. This software corresponds to a logical volume management interface in a conventional storage system.

5.3 Related Work

Systems and protocols that provide remote access to storage fall into several categories. Network-attached storage (NAS) systems provide file system-like interfaces (e.g., NFS and SMB) over the networks. These systems centralize metadata management, avoiding the need for distributed locking, but their centralized structure limits scalability.

QuickSAN most closely resembles existing SAN protocols like Fibre Channel and iSCSI. Hardware accelerators for both of these interfaces are commonly available, but iSCSI cards are particularly complex because they typically also include a TCP offload engine. Fibre Channel traditionally runs over a specialized interconnect, but new standards for Converged Enhanced Ethernet [18] (CEE) allow for Fibre Channel over lossless Ethernet (FCoE) as well. QuickSAN uses this interconnect technology. ATA over Ethernet (AoE) [41] is a simple SAN protocol designed specifically for SATA devices and smaller systems.

Systems typically combine SAN storage with a distributed, shared-disk file system that runs across multiple machines and provides a single consistent file system view of stored data. Shared-disk file systems allow concurrent access to a shared block-based storage system by coordinating which servers are allowed to access particular ranges of blocks. Examples include the Global File System (GFS2) [94], General Parallel File System [88], Oracle Cluster File System [71], Clustered XFS [91], and VxFS [103].

Parallel file systems (e.g., Google FS [33], Hadoop Distributed File System [93], Lustre [51], and Parallel NFS [39]) also run across multiple nodes, but they spread data across the nodes as well. This gives applications running at a particular node faster access to local storage.

Although QuickSAN uses existing shared-disk file systems, it actually straddles the shared-disk and parallel file system paradigms depending on the configuration. On one hand, the networked QuickSAN SSDs nodes appear as a single, distributed storage device that fits into the shared disk paradigm. On the other, each host will have a local QuickSAN SSD, similar to the local storage in a parallel file system. In both cases, QuickSAN can eliminate the software overhead associated with accessing remote storage.

Parallel NFS (pNFS) [39] is a potential target for QuickSAN, since it allows clients to bypass the central server for data accesses. This is a natural match for QuickSAN direct access capabilities.

QuickSAN also borrows some ideas from remote DMA (RDMA) systems. RDMA allows one machine to copy the contents of another machine's main memory directly over the network using a network protocol that supports RDMA (e.g., Infiniband). Previous work leveraged RDMA to reduce processor and memory IO load and improve performance in network file systems [11, 113]. RamCloud [72] utilizes RDMA to reduce remote access to DRAM used as storage. QuickSAN extends the notion of RDMA to storage and realizes many of the same benefits, especially in configurations that can utilize its userspace interface.

QuickSAN's direct-access capabilities are similar to those provided by Network Attached Secure Disks (NASD) [34]. NASD integrated SAN-like capabilities into storage devices directly exposed them over the network. QuickSAN extends this notion by also providing fast access to the local host to fully leverage the performance of solid state storage.

Alternate approaches to tackling large-scale data problems have also been proposed. Active Storage Fabrics [28], proposes combining fast storage and compute resources at each node. Processing accelerators can then run locally within the node. QuickSAN also allows applications to take advantage of data locality, but the architecture focuses on providing low-latency access to all of the data distributed throughout the network.

5.4 Results

This section evaluates QuickSAN's latency, bandwidth, and scalability along with other aspects of its performance. We measure the cost of mirroring in QuickSAN to improve reliability, and evaluate its ability to improve the energy efficiency of storage systems. Finally, we measure the benefits that distributed QuickSAN configurations can provide for sorting, a key bottleneck in MapReduce workloads. First, however, we describe the three QuickSAN configurations we evaluated.

5.4.1 Configurations

We compare performance across both centralized and distributed SAN architectures using three different software stacks.

For the centralized topology, we attached four QuickSAN SSDs to a single host machine and expose them as a single storage device. Four clients share the resulting storage device. In the distributed case, we attach one QuickSAN SSD to each of four machines to provide a single, distributed storage device. In both cases, four clients access the device. In the distributed case, the clients also host data.

We run three different software stacks on each of these configurations.

iSCSI This software stack treats the QuickSAN SSD as a generic block device and implements all SAN functions in software. In the centralized case, Linux's clustered Logical Volume Manager (cLVM) combines the four local devices into a single logical device and exposes it as an iSCSI target. In the distributed case, each machine exports its local SSD as an iSCSI target, and cLVM combines them into a globally shared device. Client machines use iSCSI to access the device, and issue requests via system calls.

Table 5.2. Request latencies for 4 KB transfer to each of the configurations to either local or remote storage. The distributed configurations have a mix of local and remote accesses, while the centralized configurations are all remote.

	Local		Remote	
	Read	Write	Read	Write
iSCSI	92.3	111.7	296.7	236.5
QuickSAN-OS	15.8	17.5	27.0	28.6
QuickSAN-D	8.3	9.7	19.5	20.7

QuickSAN-OS The local QuickSAN device at each client (a NIC in the centralized case and an SSD in the distributed case) exposes the shared storage device as a local block device. Applications access the device through the normal system call interface.

QuickSAN-D The hardware configuration is the same as QuickSAN-OS, but applications use the userspace interface to access storage.

Our test machines are dual-socket Intel Xeon E5520-equipped servers running at 2.27 GHz. The systems run CentOS 5.8 with kernel version 2.6.18. The network switch in all experiments is a Force10 S2410 CX4 10 GBit Ethernet Switch. In experiments that include a file system, we use GFS2 [32].

5.4.2 Latency

Reducing access latency is a primary goal of QuickSAN, and Table 5.2 contains the latency measurements for all of our configurations. The measurements use a single thread issuing requests serially and report average latency.

Replacing iSCSI with QuickSAN-OS reduces remote write latency by 92% – from 296 μs to 27 μs. Savings for reads are similar. QuickSAN-D reduces latency by an additional 2%. Based on the Fibre Channel latencies in Table 5.1, we expect that the saving for QuickSAN compared to a Fibre Channel-attached QuickSAN-like SSD would be smaller – between 75 and 81%.

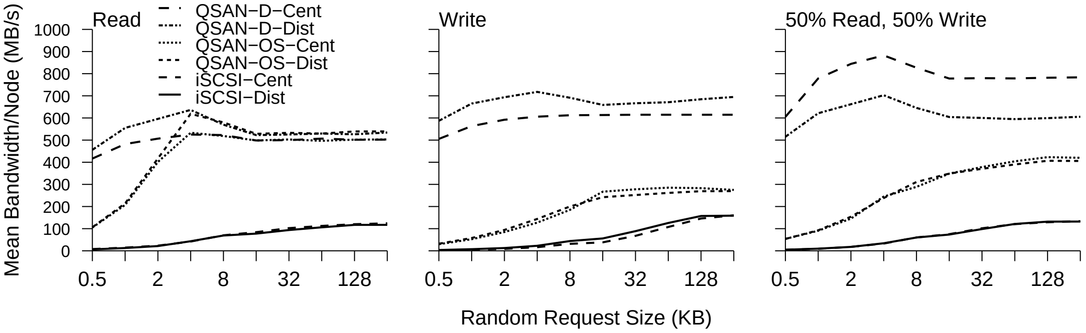

Figure 5.5. Eliminating software and block transport overheads improves bandwidth performance for all configurations. QuickSAN-D's userspace interface delivers especially large gains for small accesses and writes.

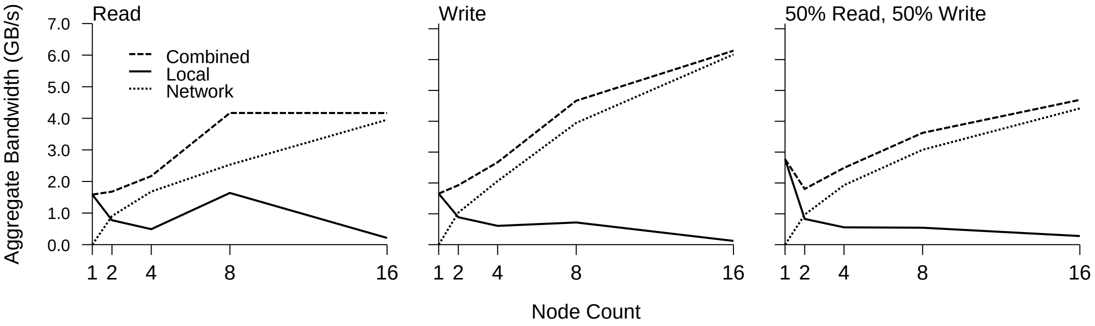

Figure 5.6. For random requests spread across the shared storage address space, aggregate bandwidth improves as more nodes are added. Local bandwidth scales more slowly since a smaller fraction of accesses go to local storage as the number of nodes grows.

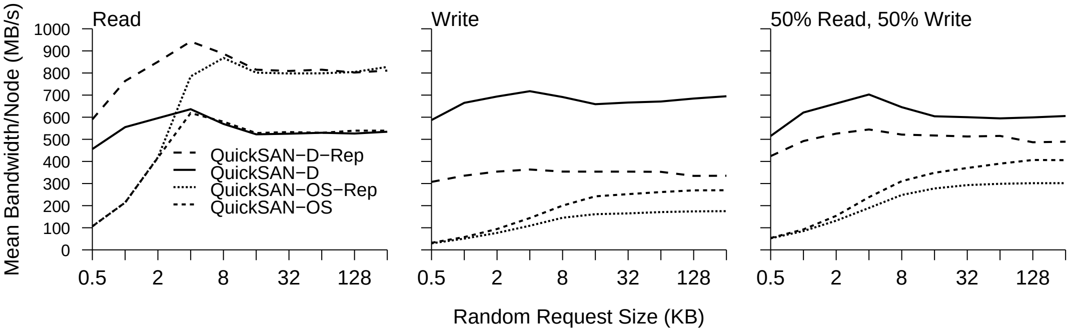

Figure 5.7. Mirroring in QuickSAN improves bandwidth for read operations because two SSDs can service any given request and, in some cases, the extra copy will be located on the faster, local SSD. Write performance drops because of the extra update operations mirroring requires.

5.4.3 Bandwidth

Figure 5.5 compares the sustained, mean per-host bandwidth for all six configurations for random accesses of size ranging from 512 bytes to 256 kB. Comparing the distributed configurations, QuickSAN-D outperforms the iSCSI baseline by 72.7× and 164.4× for small 512 B reads and writes, respectively. On larger 256 kB accesses, QuickSAN-D's advantage shrinks to 4.6× for reads and 4.4× for writes.

For QuickSAN, the distributed architecture outperforms the centralized architecture by 100 MB/s per node for 4 kB requests and maintains a 25 MB/s performance advantage across all request sizes for both kernel and userspace interfaces. Userspace write performance follows a similar trend with a 110 MB/s gain at 4 kB and at least a 45 MB/s gain for other sizes.

Write performance through the kernel is much lower due to the distributed file system overheads. Some of that cost goes towards enforcing atomicity guarantees. The performance advantage for the distributed configuration stems from having fast access to a portion of the data and from better aggregate network bandwidth to storage.

Interestingly, for a 50/50 mix of reads and writes under QuickSAN-D, the centralized architecture outperforms the distributed architecture by between 90 and 180 MB/s (17-25%) per node across all request sizes. This anomaly occurs because local and remote accesses compete for a common set of buffers in the QuickSAN SSD. This limits the SSD's ability to hide the latency of remote accesses. In the centralized case, the QuickSAN NICs dedicate all their buffer space to remote accesses, hiding more latency, and improving performance. The read/write workloads exacerbates this problem for the distributed configuration because the local host can utilize the full-duplex PCIe link, putting more pressure on local buffers than in write- or read-only cases. The random access nature of the workload also minimizes the benefits of having fast access to local storage, a key benefit of the distributed organization.

The data show peak performance at 4 kB for most configurations. This is the result of a favorable interactions between the available buffer space and network flow control requirements for transfer 4 kB and smaller. As requests get larger, they require more time to move into and out of the transfer buffers causing some contention. 4 kB appears to be the ideal request size for balancing performance buffer access between the network and the internal storage.

5.4.4 Scaling

QuickSAN is built to scale to large clusters of nodes. To evaluate its scalability we ran our 4 kB, random access benchmark on between one and sixteen nodes. There are two competing trends at work: As the number of nodes increases, the available aggregate network bandwidth increases. At the same time, the fraction of the random accesses that target a nodes local storage drops. For this study we change the QuickSAN architecture slightly to include only 16 GB of storage per node, resulting in slightly decreased per-node memory bandwidth.

Figure 5.6 illustrates both trends and plots aggregate local and remote bandwidth across the nodes as well as total bandwidth. Aggregate local bandwidth scales poorly for small node counts as more requests target remote storage. Overall, bandwidth scales well for the mixed read/write workload: Quadrupling node count from two to eight increase total bandwidth by 2.0× and network bandwidth by 3.2×. Moving from two to eight nodes increases total bandwidth by 2.5× and network bandwidth by 4.5×. Writes experience similar improvements, two to sixteen node scaling increases total bandwidth by 3.3× and network bandwidth by 5.9×. For reads, total and network bandwidth scale well from two to eight nodes (2.5× and 2.8×, respectively), but the buffer contention mentioned above negatively impacts bandwidth. Additional buffers would remedy this

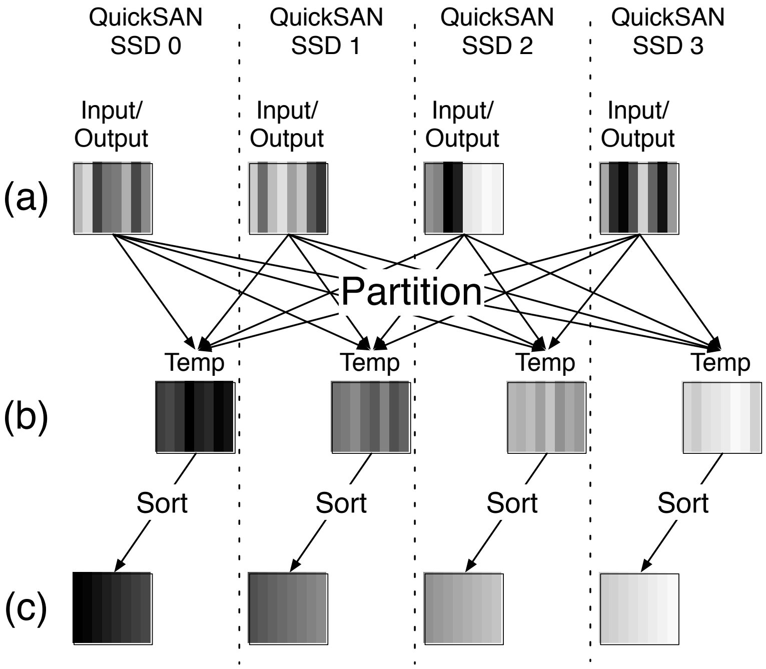

Figure 5.8. Our sorting algorithm uses a partition stage followed by parallel, local sorts. The input and output stages utilize the fast access to local storage the QuickSAN provides, while the partitioning stage leverages SSD-to-SSD communicatio.

problem. Network bandwidth continues to scale well for reads with two to sixteen node scaling producing a 4.4× increase.

Latency scaling is also an important consideration. As node counts grow, so must the number of network hops required to reach remote data. The minimum one-way latency for our network is 1 μs which includes the delay from two network interface cards, two network links, and the switch. Our measurements show that additional hops add 450 ns of latency each way in the best case.

5.4.5 Replication

SAN devices (and all high-end storage systems) provide data integrity protection in the form of simple mirroring or more complex RAID- or erasure code-based mechanisms. We have implemented mirroring in QuickSAN to improve data integrity and availability. When mirroring is enabled, QuickSAN allocates half of the storage capacity on each QuickSAN SSD as a mirror of the primary portion of another SSD. QuickSAN transparently issues two write requests for each write. QuickSAN can also select from any replicas to service remote read requests using a round-robin scheme, although it always selects a local replica, if one is available. With this scheme, QuickSAN can tolerate the failure of any one SSD.

Mirroring increases write latency by 1.7 μs for 512 B accesses and 7.7 μs for 4 KB accesses in both QuickSAN-D and QuickSAN-OS. The maximum latency impact for QuickSAN is 1.6 ×. Figure 5.7 measures the impact of mirroring on bandwidth. Sustained write bandwidth through the kernel interface drops by 23% for 4 kB requests with a maximum overhead of 35% on large transfers. The userspace interface, which has lower overheads and higher throughput, experiences bandwidth reductions of between 49 and 52% across all request sizes, but write bandwidth for 512 B requests is still 9.5 ×better than QuickSAN-OS without replication.

Replication improves read bandwidth significantly, since it spreads load more evenly across the SSDs and increases the likelihood that a copy of the data is available on the local SSD. For 4 KB accesses, performance rises by 48% for QuickSAN-D and 26% for QuickSAN-OS. Adding support for QuickSAN to route requests based on target SSD contention would improve performance further.

5.4.6 Sorting on QuickSAN

Our distributed QuickSAN organization provides non-uniform access latency depending on whether an access targets data on the local SSD or a remote SSD. By leveraging information about where data resides, distributed applications should be able to realize significant performance gains. Many parallel file systems support this kind of optimization.

To explore this possibility in QuickSAN, we implemented a distributed external sort on QuickSAN. Distributed sort is an important benchmark in part because MapReduce implementations rely on it to partition the results of the map stage. Our implementation uses the same approach as TritonSort [81], and other work [80] describes how to implement a fast MapReduce on top of an efficient, distributed sorting algorithm.

Our sort implementation operates on a single 102.4 GB file filled with key-value pairs comprised of 10-byte keys and 90-byte values. The file is partitioned across the eight QuickSAN SSDs in the system. A second, temporary file is also partitioned across the SSDs.

Figure 5.8 illustrates the algorithm. In the first stage, each node reads the values in the local slice the input file (a) and writes each of them to the portion of the temporary file that is local to the node on which that value will eventually reside (b). During the second stage, each node reads its local data into memory, sorts it, and writes the result back to the input file (c), yielding a completely sorted file. We apply several optimizations described in [81], including buffering writes to the temporary file to minimize the number of writes and using multiple, smaller partitions in the intermediate file (c) to allow the final sort to run in memory.

Figure 5.9 displays the average runtime of the partition and sort stages of our sorting implementation. The data show the benefits of the distributed storage topology and the gains that QuickSAN provides by eliminating software and block transport overheads. For distributed QuickSAN, the userspace driver improves performance 1.3× compared to the OS interface and 2.9× compared to iSCSI. Although the distributed organization improves performance for all three interfaces, QuickSAN-D sees the largest boost – 2.14× versus 1.96× for iSCSI and 1.73× for the QuickSAN-OS interface.

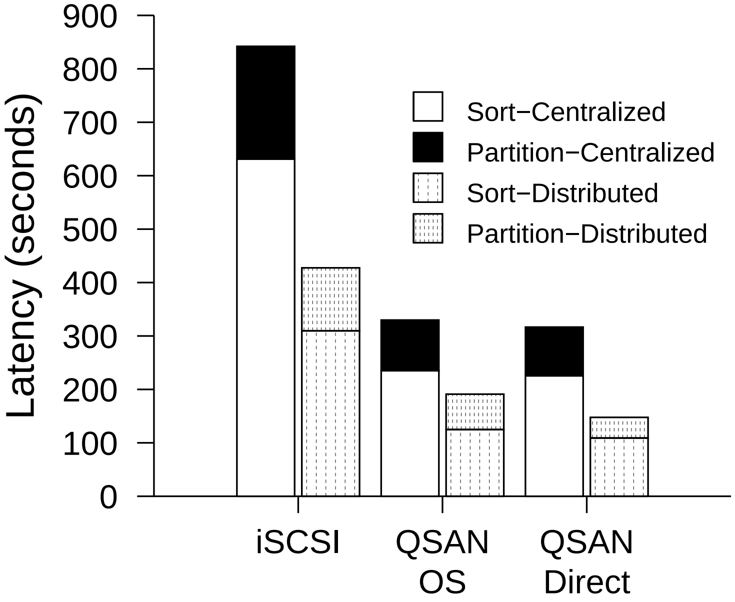

Figure 5.9. Removing iSCSI's software overheads improves performance as does exploiting fast access to local storage that a distributed organization affords.

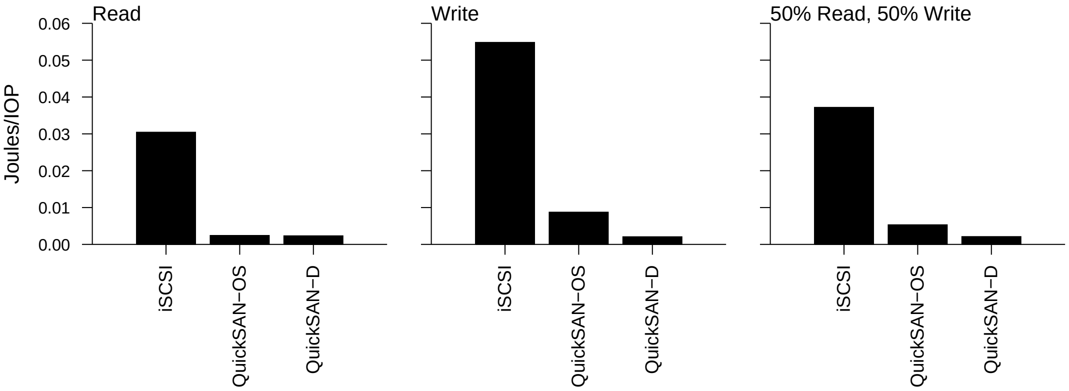

Figure 5.10. Depending on the software stack and block transport mechanism, the energy cost of storage access varies. Removing the high software overheads of iSCSI account for most of the gains, but the userspace interface saves 76% of energy for writes compared to the kernel version.

5.4.7 Energy efficiency

Since software runs on power-hungry processors, eliminating software costs for IO requests can significantly reduce the energy that each IO operation requires. Figure 5.10 plots the Joules per 4 kB IOP across the three distributed storage configurations. To collect the numbers, we measured server power at the power supply. We assume a QuickSAN SSD implemented as a product would consume 24 W (the maximum power consumed by a FusionIO ioDrive [30]), and, conservatively, that power does not vary with load. The network switch's 150 W contribution does not vary significantly with load. The trends for the centralized configurations are similar.

The graph illustrates the high cost of software-only implementations like iSCSI. The iSCSI configurations consume between 6.2 and 12.7× more energy per IOP than the QuickSAN-OS and between 13.2 and 27.4× more than QuickSAN-D.

The data for writes and the mixed read/write workload also demonstrates the energy cost of strong consistency guarantees. For writes, accessing storage through the file system increases (QuickSAN-OS) energy costs by 4× relative to QuickSAN-D.

5.4.8 Workload consolidation

An advantage of centralized SANs over distributed storage systems that spread data across many hosts is that shutting down hosts to save power does not make any data unreachable. Distributed storage systems, however, provide fast access to local data, improving performance.

The QuickSAN SSD's ability to service requests from remote nodes, even if their host machine is powered down can achieve the best of both worlds: Fast local access when the cluster is under heavy load and global data visibility to support migration.

To explore this application space, we use four servers running a persistent key-value store (MemCacheDB [17]). During normal, fully-loaded operation each server runs

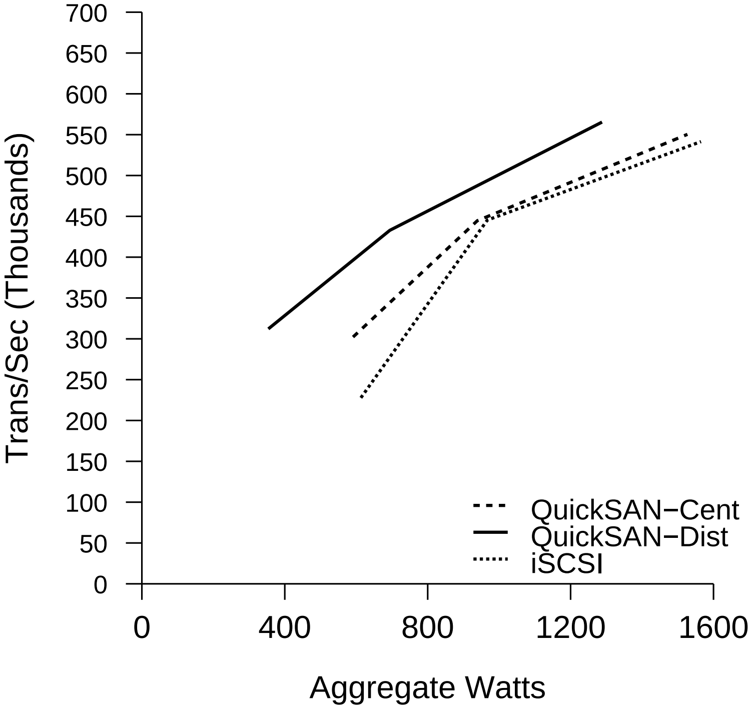

Figure 5.11. QuickSAN's ability to access remote data without host intervention allows for more efficient server consolidation. QuickSAN provides better overall efficiency with any number of servers.

a single instance of the MemCacheDB. We use four machines running memslap [65] to drive the key-value stores on the four machine under test.

As system load drops, we can migrate the key-value store to another server. On their new host, they can access the same data and transparently resume operation. We assume that a front-end steering mechanism redirects traffic as necessary using DNS. We can perform the same migration in both the centralized QuickSAN and centralized iSCSI configurations.

We measure the performance and power consumption of the system under three different configurations: centralized iSCSI, centralized QuickSAN, and distributed QuickSAN.

Figure 5.11 plots performance versus total power consumption for each system. The data show that the distributed QuickSAN reduces the energy cost of each request by 58 and 42% relative to the iSCSI and centralized QuickSAN implementations, respectively.

5.5 Summary

We have described QuickSAN a new SAN architecture designed for solid state memories. QuickSAN integrates network functionality into a high-performance SSD to allow access to remote data without operating system intervention. QuickSAN reduces software and block transport overheads by between 82 and 95% compared to Fibre Channel and iSCSI-based SAN implementations and can improve bandwidth for small requests by up to 167×. We also demonstrated that QuickSAN can improve energy efficiency by 58% compared to a iSCSI-based SAN. QuickSAN illustrates the ongoing need to redesign computer system architectures to make the best use of fast non-volatile memories.

Acknowledgements

This chapter contains material from “QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks”, by Adrian M. Caulfield and Steven Swanson, which appears in *ISCA ’13: Proceeding of the 40th Annual International Symposium on Computer Architecture*. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2013 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

Chapter 6

Summary

Emerging non-volatile memory technologies promise to significantly alter the storage performance landscape. Technologies such as phase-change memory, spin-torque transfer MRAMs, and the memristor promise orders of magnitude latency reductions and improvements in reliability over traditional hard disks and even flash memory based SSDs.

Along with these vast performance improvements, these technologies promise to dramatically alter the software landscape needed to access them efficiently. Without any changes, software overheads for accessing fast storage will quickly constitute the majority of storage request latency, approaching 97% of the total latency, and limiting the potential of these technologies.

This dissertation has focused on exploring the integration of these emerging memory technologies into computer systems through a prototype SSD called Moneta. Moneta uses a custom SSD architecture to expose fast non-volatile memories to a host system with minimal hardware overheads. Moneta's architecture is implemented in a fully functional system using an FPGA platform. Future non-volatile memory technologies are emulated using DRAM and modified memory controllers. Through this platform we can study the effects of fast storage on the whole system from hardware to applications.

Moneta gives us the opportunity to explore the correct way of redesigning the storage software stack to expose fast storage technologies without squandering their potential. Decades of system design centered on slow storage devices like disk drives has lead to thick abstraction and optimization layers in the IO path. These layers attempt to hide long storage latencies by re-ordering and scheduling IO operations. With fast, low-latency, high-throughput storage devices these software layers present significant bottlenecks to achieving the best performance from the storage array.

This work has characterized the existing IO stack and explored a series of optimizations to the SSD architecture and software stack that help eliminate these bottlenecks. We first found ways of reducing bottlenecks within the operating system IO stack by eliminating the IO scheduler and bottlenecks to concurrent access to the SSD.

Moneta's flexible prototype hardware also gives us the flexibility to study different memory latencies. Our exploration of different latencies shows that optimizing these technologies for storage applications requires a different set of trade-offs than optimizing it as a main memory replacement. We showed that Moneta's parallelism allows it to effectively hide longer access latencies, allowing for different optimizations in the memory technologies.

Optimizations to Moneta's hardware and software reduce software overheads by 62% for 4 KB operations, and enable sustained performance of 1.1M 512-byte IOPS and 541K 4 KB IOPS with a maximum sustained bandwidth of 2.8 GB/s. Moneta's optimized IO stack completes a single 512-byte IOP in 9 μs. Moneta speeds up a range of file system, paging, and database workloads by up to 8.7× compared to a state-of-the-art flash-based SSD with harmonic mean of 2.1×, while consuming a maximum power of 3.2 W.

Although, the in-kernel optimizations discussed for Moneta in Chapter 3 offer significant benefits, file system and operating system protection checks still limit performance by as much as 85%. Using Moneta-Direct, a second iteration of the Moneta prototype, we refactor the IO stack and provide a method for completely bypassing the operating system and file system on most accesses, without giving up on the sharing and protection features they provide. By allowing applications to directly talk to the hardware, IO latency is reduced by a further 64%. In addition, device throughput for small IO requests increases from 1.1M operations per seconds to 1.8M.

Moneta-Direct explores several completion techniques for notifying applications that requests have completed. These include traditional kernel interrupts and direct DMA updates to application-mapped memory regions. We show that for small requests up to 8 KB, DMA completions provide up to 171% more throughput than interrupts. However, for larger requests, issuing requests from the application and then calling the kernel and context-switching until a completion interrupt arrives provides up to 7× better bandwidth per CPU.

Moneta-Direct also provided an opportunity to implement alternative interfaces for applications to issue and complete IO requests. Using a specially designed asynchronous IO interface we showed that application level speedups of up to 1.4× are possible if users are willing to modify their applications. The asynchronous interface can also provide efficiency gains of 2.8× for a single thread in terms of CPU time used per request.

Moneta-Direct shows the benefits of carefully rethinking the IO stack across all of the system from application to hardware. By refactoring permissions and sharing code from the trusted kernel space into the hardware, Moneta-Direct exposes a much leaner interface to applications and gains significant performance benefits as well as increased interface flexibility.

In Chapter 5 we explore the implications of fast non-volatile storage in the construction of distributed storage systems. Traditionally, remote storage involves the use of a separate IO stack and network protocol overheads. Using QuickSAN, a version of the Moneta SSD with a network interface attached directly to the SSD, we distribute storage throughout the network. This allows applications to continue to take advantage of the operating system bypass optimizations, while also exploiting the data locality that comes with having a section of the storage located at each node. The direct network interface allows local and remote storage to appear as one unified address space, and remote accesses have little extra overhead beyond the additional network latency to reach the remote node.

Overall, Moneta and its variations allow us to explore the impact of fast non-volatile storage technologies before they become widely available. Using our prototype SSD we have been able to carefully examine the existing IO stack. In doing so, we have uncovered and proposed many design alternatives that will help to eliminate existing storage stack bottlenecks that prevent the best use of these emerging technologies.

Acknowledgements

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, which appears in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (MICRO '43). The dissertation author was the primary 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 “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 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. The dissertation author was the primary 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 “QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks”, by Adrian M. Caulfield and Steven Swanson, which appears in *ISCA ’13: Proceeding of the 40th Annual International Symposium on Computer Architecture*. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2013 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

Appendix A

Moneta Hardware

The Moneta hardware platform implements a high-performance SSD architecture targeting next-generation non-volatile memories. These technologies demand high-throughput, low-latency data paths to avoid squandering their performance potential. Moneta's architecture consists of several main components: a PCIe interface with associated registers and DMA controller, the request pipeline, a ring based network, and banks of memory controllers. This appendix explores each of these components and the hardware platform as a whole.

This dissertation refers to these components and describes their roles and integration with the whole system in more detail. These descriptions attempt to fill in more of the technical design of the SSD architecture.

A.1 Moneta Overview

Figure A.1 shows the high-level architecture of the Moneta storage array. Moneta's architecture provides low-latency access to a large amount of non-volatile memory spread across eight memory controllers. A scheduler manages the entire array by coordinating data transfers over the PCIe interface and the ring-based network that connects the independent memory controllers to a set of input/output queues. Moneta attaches to the computer system via an eight-lane PCIe 1.1 interface that provides a 2 GB/s full-duplex

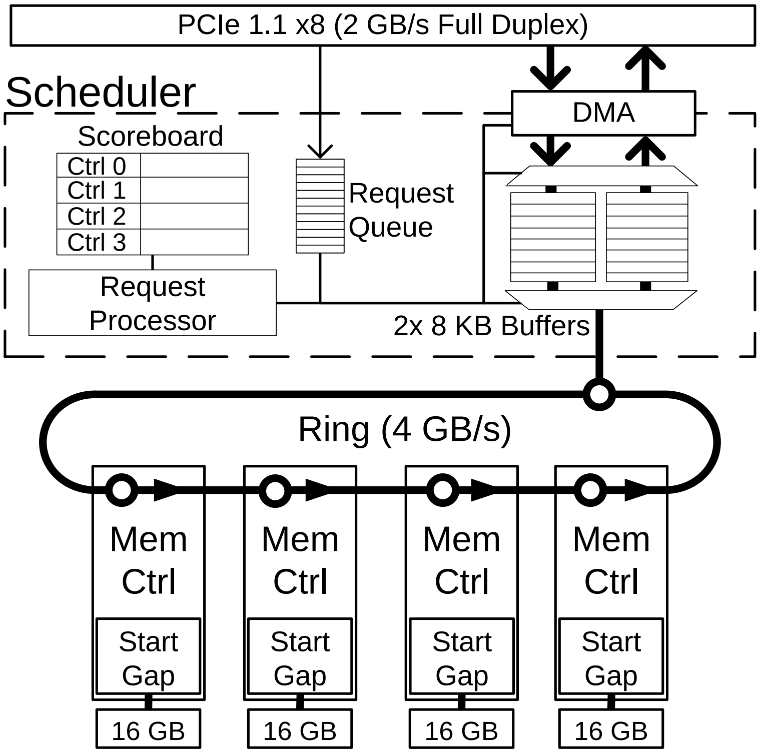

Figure A.1. Moneta's PCM memory controllers connect to the scheduler via a 4 GB/s ring. A 2 GB/s full duplex PCIe link connects the scheduler and DMA engine to the host. The scheduler manages 8 KB buffers as it processes IO requests in FIFO order. A scoreboard tracks the state of the memory controllers.

Table A.1. Moneta exposes a number of registers on the PCI-express bus to allow the host to configure the device and issue and complete I/O requests. This table shows the registers accessible on the trusted channel 0 interface.

<table>
<tr><th>0x7</th><th>0x6</th><th>0x5</th><th>0x4</th><th>0x3</th><th>0x2</th><th>0x1</th><th>0x0</th><th></th></tr>
<tr><td colspan="4">BUILDTIMESTAMP</td><td>VERS</td><td>▨</td><td colspan="2">SVNREVISION</td><td>0x0010</td></tr>
<tr><td colspan="4">▨</td><td colspan="4">CAPABILITIES</td><td>0x0018</td></tr>
<tr><td colspan="4">▨</td><td colspan="2">TAGCOUNT</td><td colspan="2">CHANCOUNT</td><td>0x0020</td></tr>
<tr><td colspan="8">CAPACITY</td><td>0x0028</td></tr>
<tr><td colspan="8">PROTECTION/ERROR</td><td>0x0038</td></tr>
<tr><td colspan="8">CHANNELSTATUS0</td><td>0x0048</td></tr>
<tr><td colspan="8">CHANNELSTATUS1</td><td>0x0050</td></tr>
<tr><td colspan="3">LENGTH</td><td colspan="5">INTADDR</td><td>0x0080</td></tr>
<tr><td colspan="4">▨</td><td>CMD</td><td>TAG</td><td>▨</td><td>DMA</td><td>0x0088</td></tr>
<tr><td>▨</td><td>FPGA</td><td colspan="6">MACADDRESS</td><td>0x0090</td></tr>
<tr><td colspan="4">LOCALREQUESTMASK</td><td colspan="4">LOCALREQUESTTAG</td><td>0x0098</td></tr>
<tr><td colspan="8">MACTABLECOMMAND</td><td>0x00A0</td></tr>
<tr><td colspan="8">CHANNELBASE0</td><td>0x0800</td></tr>
<tr><td colspan="8">CHANNELBASE1</td><td>0x0808</td></tr>
<tr><td colspan="8">CHANNELBASE2</td><td>0x0810</td></tr>
<tr><td colspan="8">...</td><td></td></tr>
<tr><td colspan="8">CHANNELBASE255</td><td>0x0FF8</td></tr>
<tr><td colspan="8">TAGSTATUS0</td><td>0x1000</td></tr>
<tr><td colspan="8">TAGSTATUS1</td><td>0x1008</td></tr>
<tr><td colspan="8">TAGSTATUS2</td><td>0x1010</td></tr>
<tr><td colspan="8">...</td><td></td></tr>
<tr><td colspan="8">TAGSTATUS255</td><td>0x17F8</td></tr>
</table>

connection (4 GB/s total).

A.1.1 Registers

Moneta exposes a set of registers that the host uses to configure the device and issue commands for the hardware to process. Table A.1 shows the register layout for the first 4 KB of address space exposed by the device. The remaining space replicates the layout depicted in Table A.2 for every 4 KB range. These replicated registers have identical functionality, but can be mapped directly into application address spaces - one

Table A.2. Moneta exposes a number of registers on the PCI-express bus to allow the host to configure the device and issue and complete I/O requests. This table shows the registers accessible on the untrusted channels.

0x7	0x6	0x5	0x4	0x3	0x2	0x1	0x0	
LENGTH			INTADDR					0x0080
				CMD	TAG		DMA	0x0088

4 KB replica per application. By using unique addresses for each application, we can use the TLB in the processor to guarantee that the written address uniquely identifies the application.

There are three main register groups. The request issuing registers, configuration registers and request status registers.

The registers for issuing requests capture information about each I/O request and are written in a single atomic operation by the host's CPU. Commands are 16 bytes in length and contain the following set of information: an identifying tag (*TAG*), a destination address (*INTADDR*), a length in bytes (*LENGTH*), a command (*CMD*), and an indicator of the request method for completing the request (*DMA*).

Configuration registers allow the host system to configure Moneta for general use. These include a set of read-only information registers:

1. *BUILDTIMESTAMP*, *VERS*, and *SVNREVISION* give general information about the system date/time when the hardware was build, what version of the hardware, and the source code control revision number at the time of the build, respectively.

2. *CAPABILITIES* provides a bit-vector of features that the hardware build provides. Possible features include networking and virtualization support.

3. *TAGCOUNT* and *CHANCOUNT* signal to the host the number of outstanding requests the hardware can track at once per channel, and the number simultaneous channels, or virtual interfaces, the hardware can support.

4. *CAPACITY* lists the capacity of the SSD in bytes.

5. *FPGA* gives the current FPGA id that the hardware is running on

6. *MACADDRESS* gives the current device ethernet MAC address

The following read/write registers configure the device for use:

1. *PROTECTION* is a write-only 64-bit register used by the driver to issue commands to the protection management block. The commands are described in that section.

2. *ERROR* is a read-only 64-bit register that reads the first element off an error queue to communicate errors in requests back to the host system.

3. *LOCALREQUESTMASK*, in combination with *LOCALREQUESTTAG*, is used by the hardware to identify which requests belong to this device, and which requests should be handled by other devices in the storage area network. *LOCALREQUESTMASK* is bitwise-ANDed with the upper bits of *INTADDR* when a request is received. If the result matches the value in *LOCALREQUESTTAG*, the request is processed lcoally.

4. *MACTABLECOMMAND* is used to issue commands to the network interface to setup the list of additional devices that make up a storage area network.

5. *CHANNELBASE#* stores a pointer to the start of a contiguous region of host memory to be used by this channel. Each channel has its own base pointer. The memory region contains both DMA buffers and control pages used for communication between the device and host.

Finally, the remaining registers are used during the completion phase of each request. These registers are:

1. *CHANNELSTATUS#* store bit-vectors, where each bit in these registers represent a single channel. If the bit is set, that channel has completed one or more requests since the last read from this register. Reads atomically reset all of the bits in this register to zero. The driver will read these registers to determine which of the *TAGSTATUS#* registers to read.

2. *TAGSTATUS#* store bit-vectors, where each bit in these registers represent a single tag in the relevant channel. Each bit is set when a request with the correct channel and tag complete and issue an interrupt. Reads atomically reset all of the bits in this register to zero.

A.1.2 DMA

The Moneta DMA controller interfaces between the buffers in the request pipeline and an instance of the Xilinx PCIe endpoint core [111]. It can split large transfers into pieces to meet the various PCIe 1.1 specification requirements [75]. The DMA engine can issue an unlimited number of transfers to the host system at once and can track 8 outstanding transfers from the host. The disparity occurs because PCIe transfers to the host do not require responses, and thus no state must be tracked for those request.

To ensure that the link is fully utilized, the DMA engine is designed to never block incoming data. This requires that the client interface must be ready to accept data as soon as it is available. The controller must (and clients should) also be capable of both receiving and transmitting data at the same time to fully saturate the PCIe link. Moneta is capable of this and can transfer more than 3 GB/s when performing both reads and writes at the same time, even though the link provides only 2 GB/s of bandwidth in each direction.

When the host system boots, the PCIe endpoint core and the host negotiate address ranges and maximum request and packet sizes to be used by the clients. The

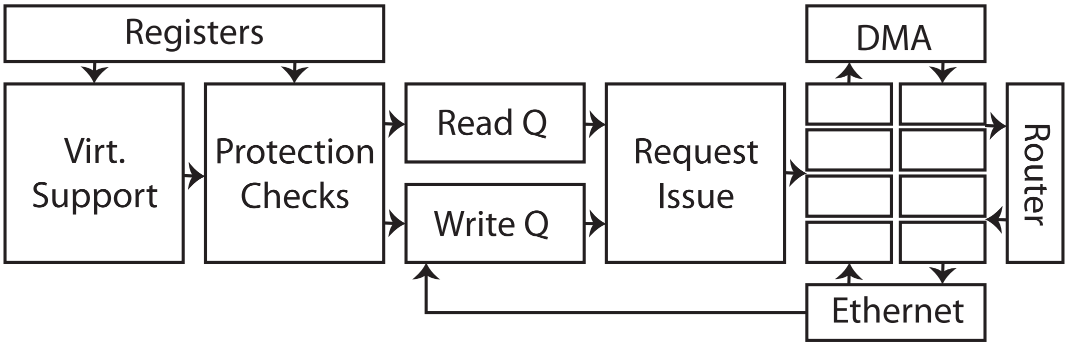

Figure A.2. The Request Pipeline receives requests from the registers and passes them through a number of state machines that support virtualization, protection checking, and request execution functions. Once issued, requests are processed and transferred in the request buffers between the host via DMA, memory controllers through the local router, or a remote device through Ethernet.

DMA controller uses these parameters to break up DMA transfers into valid sizes before issuing them to the host. Individual responses can result in responses returning out of order, and the client should be able to handle this. The DMA controller calculates the offset that each response belongs at, and provides this information to the client along with the data.

The host systems currently in use with Moneta always return data in-order and so we simplify the buffer logic by using FIFOs instead of fully addressable rams to store the incoming data from DMA transfers.

A.2 Request Pipeline

Figure A.2 shows the components that, together, form the request pipeline. Requests arrive from the host over PCIe as PIO writes and are decoded by the registers. The "Virtualization Support" block provides the hardware support necessary to allow applications to directly talk to the hardware, bypassing the operating system, as described in Chapter 4. The "Protection" block implements basic access control to limit what subset of the device requests can access. The "Request Queues" implement one or more

FIFOs to sort and store requests until they are processed.

A.2.1 Virtualization

The virtualization support block has one main function: to assign and translate between physical tags and virtual channel and tag pairs. The Moneta hardware can track 64 in-flight operations in the hardware at once. Many more can exist in the hardware queues written by the virtual channels. Limiting the number of in-flight requests reduces the number of bits of data dedicated to tracking which request is which throughout the rest of the system.

By locating this translation early in the request pipeline, the remainder of the pipeline can choose to store only the physical tag, which requires just 6 bits instead of 14-16 depending on the number of supported virtual channels.

As requests pass through this block, the virtual channel and tag are recorded in a table and a physical tag is assigned. The request will own that tag until it completes - either successfully or in error. Once the request finishes, the physical tag is freed and can be allocated to a waiting request.

A.2.2 Protection

Moneta implements a hardware protection block to verify that each request accesses only the set of blocks the issuing process and channel are permitted to access. To facilitate this, the protection checking block contains a memory buffer that stores 16 thousand extent-based protection records. These records contain pointers to other records, through which a balanced binary tree can be built.

A set of root pointers, one per channel, locates the top of the binary tree of records for each channel.

Using the channel number provided by the registers and virtualization block, the protection block first looks up the head pointer for the given channel. It then begins comparing the range of blocks requested to the *START* and *LEN* fields of each record in the tree to determine if the supplied request is permitted or not.

Each verification step takes 3 cycles (12 ns). This includes reading a record from memory and matching it against the request. Since the tree is maintained with a balanced layout, the maximum depth of the 16 K entry tree is 14, making the maximum lookup time 168 ns.

Once a request has been validated, it gets passed on to the next block in the request pipeline.

Failed requests generate an error, which gets passed back up to the application and operating system for handling. Requests can fail in two ways. The first is a missing permission record, in which no channel and address combination matched the given request. The second is a permission failure, which occurs when a record exists covering the requested address range, but does not provide the necessary permission for the request (i.e. write request received, but only read permissions granted). Applications and the operating system can choose to handle these cases differently. The software level is discussed in more detail in the rest of this dissertation.

A.2.3 Request Queues

There are two request queues between the protection block and the request issue and buffers. These queues hold read and write requests separately so that when requests are issued Moneta can alternate between read and write requests. By alternating, the DMA engine will handle roughly half reads and half writes, maximizing the throughput achievable on the PCIe link.

Moneta can also be configured to use a single queue holding both reads and writes when ordering constraints are more important than throughput. In this case requests are issued in order, as they are received over the PCIe link.

A.2.4 Transfer Buffers and Scheduler

The transfer buffers and scheduler control logic are the main state machines for request processing. The scheduler orchestrates Moneta's operation, assigning requests to buffers and generating the accesses to the DMA controller and memory controllers over the ring network.

Moneta includes up to 8 buffers, each able to hold an 8 KB segment of a request. The size was picked to match the memory buffer sizes and to facilitate striping of data across multiple memory controllers.

Requests are split into segments of up to 8 KB in size, or whenever an 8 KB boundary would be crossed. By splitting at these points, the request both fit in the available buffer and will be directed at a single memory controller.

Both read and write requests are allocated buffers immediately. Write requests generate a request to the DMA controller (described earlier), and wait for the data to return before proceeding. Read requests perform a memory access first, with the DMA request generated once the data is received from the ring or ethernet networks.

A.3 Host-Interface

The host-interface defines the set of interfaces that the host system and the Moneta hardware use to communicate. This includes a set of registers that the kernel and applications can read and write to configure the hardware and issue and complete I/O requests. Moneta uses a PCI-express interface to connect to the host. This link uses the PCIe 1.1 specification with eight lanes, providing 2 GB/s of bandwidth in each direction between then host and the hardware. The link runs at full duplex, so the total effective bandwidth is 4 GB/s when reads and writes occur at the same time.

A.3.1 Completing Requests in Hardware

Throughout the development of Moneta, we have explored several different techniques for completing requests. At the hardware level, these translate into two distinct mechanisms. The first is the traditional hardware interrupt. Moneta raises an interrupt through the PCIe core which causes the operating system to run an interrupt handler. The second is writing a value using DMA transfers to a known memory location. In this case, the software must periodically check for a change in value at that location. DMA completions are useful because they allow the hardware to communicate to an application without operating system involvement (if the physical address is mapped into that application's address space).

We cover completions in more detail when discussing user-space access and device virtualization in Chapter 4.

A.4 Ring Network

Moneta uses a 128-bit wide data path to move data between its memory controllers and transfer buffers. The data path is organized into a ring topology, with an instance of the router shown in Figure A.3 interfacing between the ring and the various client components. The figure shows the router interfaces and the data signal with a mux controlling the source of the output data. The other output signals would also have similar muxes. Only one component can transmit data onto the ring at a time, with ownership controlled by a token. When the token is held, the muxes on the output pins direct the source of data to come from the client TX port. Routers release the token at the conclusion of each packet, ensuring that all transmitters are given a chance to make forward progress.

Data moves around the ring at 250 MHz. The 128-bit data width at 250 MHz

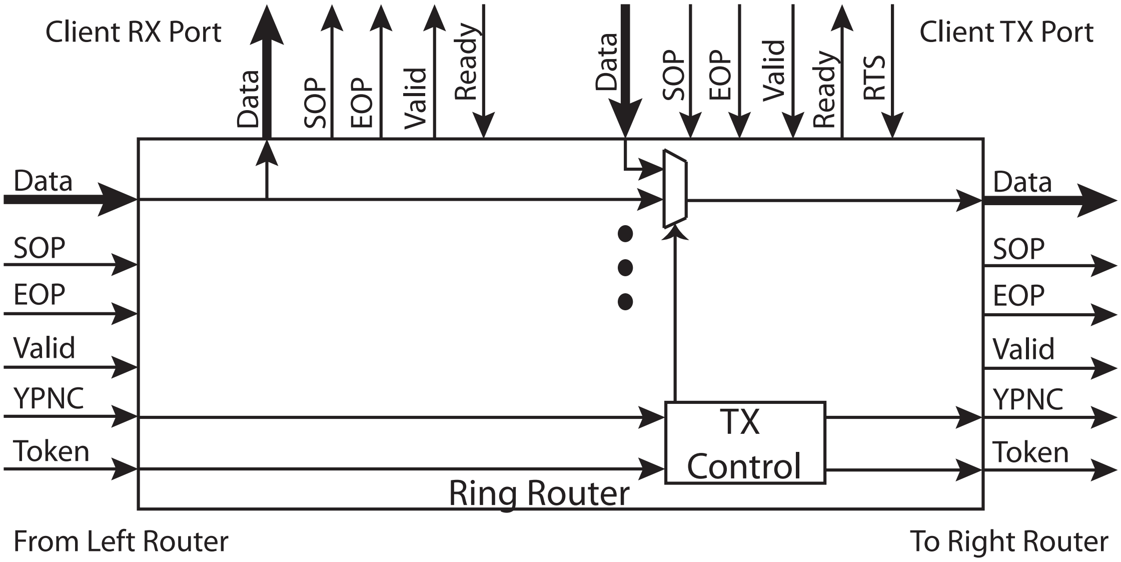

Figure A.3. The router provides an interface to the ring network for the request pipeline and the memory controllers. It has input and output ports for the ring interface which connect to other routers and the client port, which provides a input/output interface to send and receive packets.

translates to a maximum bandwidth of 4 GB/s.

The routers implement a basic form of flow-control, by allowing the receiving router to leave flits of data on the ring. When a receiving router is unable to read data from the network, it sets the YPNC flag and leaves the data circulating on the network. When the transmitting router notices an incoming flit of data with the YPNC flag set, it halts transmission and passes the YPNC flit on. In this way, the ring is used as a buffer for all of the unread data. When the destination is again ready to accept data, it waits for the YPNC flag to arrive on its input port and begins removing flits from the ring. When the original transmitter sees a flit with no valid data it knows it can once again resume transmitting.

A.5 Memory Controllers

Each of Moneta's memory controllers manages an independent bank of memory. The controllers connect to the scheduler via the ring network and provide a pair of 8 KB queues to buffer incoming and outgoing data. The memory array at each controller comprises two DIMM-like memory modules that present a 72 bit (64 + 8 for ECC) interface.

Each DIMM contains four internal banks and two ranks, each with an internal row buffer. The banks and their row buffers are 8 KB wide, and the DIMM reads an entire row from the memory array into the row buffer. Once that data is in the row buffer, the memory controller can access it at 250 MHz DDR (500M transfers per second), so the peak bandwidth for a single controller is 4 GB/s.

The memory controller implements the start-gap wear-leveling and address randomization scheme [76] to evenly distribute wear across the memory it manages.

Moneta is designed to study emerging memory technologies that are not yet commercially available or have not yet reached their projected performance targets. To facilitate this, an emulation layer allows Moneta to slow down regular DDR2 DRAM to match the performance characteristics of future technologies. This is described in more detail in the next section.

A.5.1 Non-Volatile Memory Emulation

The Moneta prototype implements an emulation layer that slows down DRAM access latencies to match the performance targets of new non-volatile storage technologies, in real-time. This emulation layer provides two benefits. First, the memory emulation gives us a head-start on designing systems for these fast memories. By running complete systems in real-time we are able to run whole applications to completion, rather than microbenchmarks or whole-system simulations that only cover a small fraction of an application's execution. At the same time, we can examine all of the impacts that fast non-volatile memories will have on storage and system performance.

Second, emulating non-volatile memory in real-time allows us to explore the impact that different memory latencies will have on the overall system performance. This allows us to explore how sensitive system performance is to an emerging technology missing its projected performance numbers.

The BEE3 [7] platform contains the physical wiring to support 2 banks of DDR2 DRAM on each of its four Xilinx FPGAs. Moneta populates these banks with 2 4 GB DIMMs on each bank which connect to an instantiated Xilinx memory interface core. In total these DIMMs provide 64 GB of storage in each Moneta device – 16 GB on each FPGA.

Moneta's memory controllers emulate non-volatile devices on top of DRAM using a modified version of the Xilinx Memory Interface Generator DDR2 controller. It adds latency between the read address strobe and column address strobe commands during reads and extends the precharge latency after a write. The controller can vary the apparent latencies for accesses to memory from 4 ns to over 500 μs. We use the values from [53] (48 ns and 150 ns for array reads and writes, respectively) to model PCM in this work, unless otherwise stated.

The width of memory arrays and the corresponding row buffers are important factors in the performance and energy efficiency of PCM memory [53]. The memory controller can vary the effective width of the arrays by defining a virtual row size and inserting latencies to model opening and closing rows of that size. The baseline configuration uses 8 KB rows.

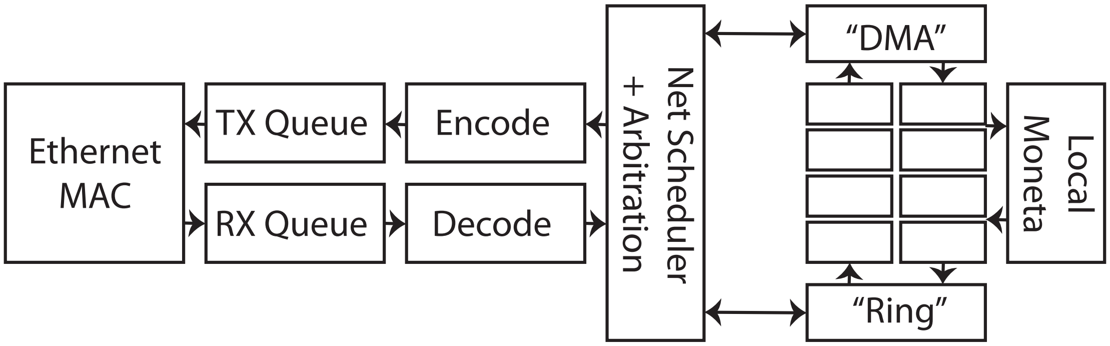

Figure A.4. The network interface includes an Ethernet MAC core, packet buffers, and the network state machine. The state machine processes incoming requests and wraps outgoing requests in ethernet frames a custom low level protocol.

A.6 Ethernet Network

The inclusion of an ethernet network interface in the Moneta hardware provides the mechanism for connecting multiple Moneta devices together on a network and allowing multiple hosts to share the same storage device. Figure A.4 shows the network interface flow and its interfaces to the existing buffers and the rest of the Moneta design.

Moneta uses a 10 Gb ethernet interface with passive, copper CX4 connections. The network state machine interfaces to the request buffers and to the request queues.

As packets arrive from the ethernet interface, they are written into the receive packet buffer. This buffer has room for 4096 8-byte words, or 32 KB of data. Packets are written into the buffer completely before the next state machine is notified that any data has arrived. By delaying the start of processing until the whole packet is received, corrupt and invalid packets can be safely dropped. The ethernet frame checksum is used to verify the validity of the packet and payload. The downside is the introduction of slightly longer latencies. Writing an 8 KB packet into the buffer takes 1024 clock cycles, at ethernet line rate of 125 MHz that means an additional 8192 ns.

The receive buffer capacity was chosen to hold several maximum sized packets and minimize the amount of flow control needed to prevent packet loss. When the receive buffer reaches one-quarter full the flow control state machine kicks in and issues pause requests to the ethernet MAC core. Once the buffer falls below an eighth full, the pause requests are cancelled. The threshold is set relatively low because several packets may be in transit before the pause request is processed by the device at the other end of the link.

Received packets exit the packet buffer and the receive state machine processes them. There are only four types of packets that the state machine recognizes and the packet formats are shown in Table A.3. There are two request formats, for reads and writes, and two completion formats, for signalling that read and write requests have finished. The request headers are nearly identical and contain the normal source and destination MAC addresses, we also include a Source Index which is just a lookup into the preconfigured, identical MAC address tables on each node. A tag is set along with the request so the origin can identify the completion when it arrives. The request also includes a request length, storage address, and a few other flags that Moneta uses during request processing. These fields almost map directly onto the existing fields in a local-only version of Moneta. The Source Index is split across the VirtualTag and ChannelID as these fields don't change during the processing of the requests locally.

Once the type of packet is identified, the state machine branches to a handler for that packet type. Read requests are simply output into the request queue as they have no payload. Write requests are also output, but the state machine will wait for a buffer allocation for the request and then transferring the data from the packet buffer into the main request buffer. Completions are handled in a similar manner, read completions contain a payload which is written directly into the buffer (allocated during the initial read request), write completions require no buffer and simply use the normal mechanisms to signal that the request is complete.

To interface the network with the rest of Moneta, the network controller emulates

Table A.3. Moneta uses raw ethernet frames with minimal overhead to communicate between nodes on the network. This table shows the general format of these packets. This packet represents a write request and includes a data payload. Read requests, simply change the OP field and include no payload. Completion packets are also sent, these return the TAG and a data payload if necessary.

0x7	0x6	0x5	0x4	0x3	0x2	0x1	0x0	
SRC MAC		DEST MAC						0x0000
INDEX,OP[3:0]		0x70	0x88	SRC MAC				0x0008
WRITE ADDRESS								0x0010
SIZE		TAG		OFFSETS		CMD	CTRL	0x0018
PAYLOAD - WRITE DATA								0x0020

two of the existing Moneta components – the DMA controller and the Ring network. This simplifies the integration by allowing the existing scheduler and state machines to use the same interfaces for both local and remote data transfers. When a request needs to be forwarded to a remote node, instead of talking to the existing ring controller, the network "ring" interface is used. Because the interfaces are the same, just a few of the control signals need to be changed in the existing state machines. Similarly, remote requests that arrive from the network are inserted into the request queue and then a network "DMA" read is used to fetch the associated data from the network buffers.

A.7 Summary

This appendix explores the Moneta prototype SSD hardware architecture and implementation in extensive detail. The Moneta prototype targets emerging non-volatile memory technologies that have performance characteristics approaching that of DRAM. Suitable technologies include phase-change memory, spin-torque transfer memory, and the memristor. The low-latency, high-endurance characteristics of these memories make them appealing choices for very fast storage arrays like Moneta.

Moneta focuses on providing a high-throughput simple data-path to move data between the storage devices and the host machine. By ensuring adequate buffering and a high-speed network to connect the memory devices to the PCIe link, Moneta can saturate the full-duplex connection to the host network and sustain almost 2 million IO operations per second.

Acknowledgements

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, which appears in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (MICRO ’43). The dissertation author was the primary 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 “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 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. The dissertation author was the primary 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 "QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks", by Adrian M. Caulfield and Steven Swanson, which appears in *ISCA '13: Proceeding of the 40th Annual International Symposium on Computer Architecture*. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2013 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.

Bibliography

[1] The lustre project. http://www.lustre.org/.

[2] International technology roadmap for semiconductors: Emerging research devices, 2007.

[3] Maxim Adelman. Principle Engineer, Violin Memory. Personal communication.

[4] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: a Fast Array of Wimpy Nodes. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 1–14, New York, NY, USA, 2009. ACM.

[5] Hyokyung Bahn, Soyoon Lee, and Sam H. Noh. P/PA-SPTF: Parallelism-aware Request Scheduling Algorithms for MEMS-based Storage Devices. *Trans. Storage*, 5(1):1–17, 2009.

[6] 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.

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

[8] M. A. Blumrich, C. Dubnicki, E. W. Felten, and Kai Li. Protected, user-level DMA for the SHRIMP network interface. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, HPCA '96, pages 154–, Washington, DC, USA, 1996. IEEE Computer Society.

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

[10] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the hamlyn sender-managed interface architecture. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 245–259, New York, NY, USA, 1996. ACM.

[11] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. Nfs over rdma. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, NICELI '03, pages 196–208, New York, NY, USA, 2003. ACM.

[12] 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 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Adrian M. Caulfield, Todor I. Mollov, Louis Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceeding of the 17th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, March 2012. ACM.

[14] Yu-Bin Chang and Li-Pin Chang. A self-balancing striping scheme for nand-flash storage systems. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1715–1719, New York, NY, USA, 2008. ACM.

[15] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *SIGMETRICS '09: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, pages 181–192, New York, NY, USA, 2009. ACM.

[16] Shimin Chen. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 73–86, New York, NY, USA, 2009. ACM.

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

[18] Cisco. Lossless 10 gigabit ethernet: The unifying infrastructure for san and lan consolidation, 2009.

[19] Joel Coburn, Trevor Bunker, Rajesh K. Gupta, and Steven Swanson. From ARIES to MARS: Reengineering transaction management for next-generation, solid-state drives. Technical Report CS2012-0981, Department of Computer Science & Engineering, University of California, San Diego, June 2012. http://csetechrep.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2012-0981.

[20] John D. Davis and Lintao Zhang. FRP: A Nonvolatile Memory Research Platform Targeting NAND Flash. In *Proceedings of First Workshop on Integrating Solid-State Memory in the Storage Hierarchy*, 2009.

[21] Cagdas Dirik and Bruce Jacob. The Performance of PC Solid-State Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 279–289, New York, NY, USA, 2009. ACM.

[22] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 7–7, Berkeley, CA, USA, 2008. USENIX Association.

[23] Ivan Dramaliev and Tara Madhyastha. Optimizing Probe-Based Storage. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 103–114, Berkeley, CA, USA, 2003. USENIX Association.

[24] Marcus Dunn and A. L. Narasimha Reddy. A new i/o scheduler for solid state devices. Technical Report TAMU-ECE-2009-02-3, Department of Electrical and Computer Engineering Texas A&M University, 2009.

[25] Kaoutar El Maghraoui, Gokul Kandiraju, Joefon Jann, and Pratap Pattnaik. Modeling and Simulating Flash Based Solid-State Disks for Operating Systems. In *WOSP/SIPEW '10: Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 15–26, New York, NY, USA, 2010. ACM.

[26] Elpida. Elpida DDR2 SDRAM EDE1104AFSE Datasheet, 2008. http://www.elpida.com/pdfs/E1390E30.pdf.

[27] Everspin Technologies. Spin-Torque MRAM Technical Brief. http://www.everspin.com/PDF/ST-MRAM_Technical_Brief.pdf.

[28] Blake G. Fitch, Aleksandr Rayshubskiy, Michael C. Pitman, T. J. Christopher Ward, and Robert S. Germai n. Using the Active Storage Fabrics Model to Address Petascale Storage Challenges. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 47–54, New York, NY, USA, 2009. ACM.

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

[30] iodrive2 data sheet. http://www.fusionio.com/data-sheets/iodrive2/.

[31] Greg Ganger, Bruce Worthington, and Yale Patt. Disksim. http://www.pdl.cmu.edu/DiskSim/.

[32] http://sourceware.org/cluster/gfs/.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[34] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 92–103, New York, NY, USA, 1998. ACM.

[35] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGOPS Oper. Syst. Rev.*, 32:92–103, October 1998.

[36] Dror Goldenberg. Infiniband device virtualization in xen. http://www.mellanox.com/pdf/presentations/xs0106_infiniband.pdf.

[37] John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John C. Bucy, and Gregory R. Ganger. Timing-accurate Storage Emulation. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 6, Berkeley, CA, USA, 2002. USENIX Association.

[38] Jiahua He, Arun Jagatheesan, Sandeep Gupta, Jeffrey Bennett, and Allan Snavely. DASH: a Recipe for a Flash-based Data Intensive Supercomputer, 2010.

[39] Dean Hildebrand and Peter Honeyman. Exporting Storage Systems in a Scalable Manner with pNFS. In *Symposium on Mass Storage Systems*, pages 18–27, 2005.

[40] Yenpo Ho, G.M. Huang, and Peng Li. Nonvolatile Memristor Memory: Device Characteristics and Design Implications. In *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 485 –490, 2-5 2009.

[41] S. Hopkins and B. Coile. Aoe (ata over ethernet), 2009. http://support.coraid.com/documents/AoEr11.txt.

[42] Amber Huffman and Joni Clark. Serial ATA Native Command Queuing, July 2003. http://www.seagate.com/content/pdf/whitepaper/D2c_tech_paper_intc-stx_sata_ncq.pdf.

[43] IBM 3340 Disk Storage Unit. http://www-03.ibm.com/ibm/history/exhibits/storage/storage_3340.html.

[44] http://www.intel.com/content/www/us/en/solid-state-drives/ssd-910-series-specification.html.

[45] Intel. Intel system controller hub datasheet, 2008. http://download.intel.com/design/chipsets/embedded/datashts/319537.pdf.

[46] Intel X58 Express Chipset Product Brief, 2008. http://www.intel.com/products/desktop/chipsets/x58/x58-overview.htm.

[47] Intel Corporation. Intel X25-E SATA Solid State Drive Product Manual. http://download.intel.com/design/flash/nand/extreme/extreme-sata-ssd-datasheet.pdf.

[48] Silicon Graphics International. XFS: A high-performance journaling filesystem. http://oss.sgi.com/projects/xfs.

[49] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Disk schedulers for solid state drivers. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 295–304, New York, NY, USA, 2009. ACM.

[50] David Kotz, Song B Toh, and Sriram Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Technical report, Dartmouth College, Hanover, NH, USA, 1994.

[51] Petros Koutoupis. The lustre distributed filesystem. *Linux J.*, 2011(210), October 2011.

[52] Laura M. Grupp and Adrian M. Caulfield and Joel Coburn and John Davis and Steven Swanson. Beyond the Datasheet: Using Test Beds to Probe Non-Volatile Memories' Dark Secrets. In *IEEE Globecom 2010 Workshop on Application of Communication Theory to Emerging Memory Technologies (ACTEMT 2010)*, 2010.

[53] 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.

[54] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems.

[55] Eunji Lee, Kern Koh, Hyunkyoung Choi, and Hyokyung Bahn. Comparison of I/O Scheduling Algorithms for High Parallelism MEMS-based Storage Devices. In *SEPADS'09: Proceedings of the 8th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, pages 150–155, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).

[56] S. Lee, K. Fleming, J. Park, Ha K., Adrian M. Caulfield, Steven Swanson, Arvind, and J. Kim. BlueSSD: An Open Platform for Cross-layer Experiments for NAND Flash-based SSDs. In *Proceedings of the 2010 Workshop on Architectural Research Prototyping*, 2010.

[57] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1075–1086, New York, NY, USA, 2008. ACM.

[58] Joshua LeVasseur, Ramu Panayappan, Espen Skoglund, Christo du Toit, Leon Lynch, Alex Ward, Dulloor Rao, Rolf Neugebauer, and Derek McAuley. Standardized but flexible i/o for self-virtualizing devices. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 9–9, Berkeley, CA, USA, 2008. USENIX Association.

[59] Kirill Levchenko, Andreas Pitsillidis, Neha Chachra, Brandon Enright, Márk Félegyházi, Chris Grier, Tristan Halvorson, Chris Kanich, Christian Kreibich, He Liu, Damon McCoy, Nicholas Weaver, Vern Paxson, Geoffrey M. Voelker, and Stefan Savage. Click Trajectories: End-to-End Analysis of the Spam Value Chain. In *Proceedings of the IEEE Symposium and Security and Privacy*, Oakland, CA, May 2011.

[60] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[61] Jiuxing Liu and Bulent Abali. Virtualization polling engine (vpe): using dedicated cpu cores to accelerate i/o virtualization. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 225–234, New York, NY, USA, 2009. ACM.

[62] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[63] K. Magoutis, M. Seltzer, and E. Gabber. The case against user-level networking. In *Proceedings of the Workshop on Novel Uses of System-Area Networks*, SAN-3, 2004.

[64] Derek McAuley and Rolf Neugebauer. A case for virtual channel processors. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, NICELI '03, pages 237–242, New York, NY, USA, 2003. ACM.

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

[66] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[67] Young Jin Nam and Chanik Park. Design and Evaluation of an Efficient Proportional-share Disk Scheduling Algorithm. *Future Gener. Comput. Syst.*, 22(5):601–610, 2006.

[68] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*, pages 145–158, New York, NY, USA, 2009. ACM.

[69] Spencer W. Ng. Improving Disk Performance Via Latency Reduction. *IEEE Trans. Comput.*, 40(1):22–30, 1991.

[70] Numonyx. Numonyx Omneo P8P PCM 128-Mbit Parallel Phase Change Memory Datahseet, April 2010. http://numonyx.com/Documents/Datasheets/316144_P8P_DS.pdf.

[71] https://oss.oracle.com/projects/ocfs2/.

[72] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[73] R. Osborne, Q. Zheng, J. Howard, R. Casley, D. Hahn, and T. Nakabayashi. Dart – a low overhead atm network interface chip. In *Proceedings of the 1996 4th IEEE Symposium on High Performance Interconnects*, pages 175–186, 1996.

[74] Pci-sig - i/o virtualization. http://www.pcisig.com/specifications/iov/.

[75] Pci-sig - pcie base specification 1.1. http://www.pcisig.com/specifications/pciexpress/base.

[76] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Last ras, 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.

[77] 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.

[78] Himanshu Raj and Karsten Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC '07, pages 179–188, New York, NY, USA, 2007. ACM.

[79] The ramp project. http://ramp.eecs.berkeley.edu/index.php?index.

[80] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: an i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 13:1–13:14, New York, NY, USA, 2012. ACM.

[81] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. Tritonsort: a balanced large-scale sorting system. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[82] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38:39–47, May 2005.

[83] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 27(3):17–28, 1994.

[84] Julian Satran, Leah Shalev, Muli Ben-Yehuda, and Zorik Machulsky. Scalable i/o - a well-architected way to do scalable, secure and virtualized i/o. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 3–3, Berkeley, CA, USA, 2008. USENIX Association.

[85] Lambert Schaelicke and Al Davis. Improving i/o performance with a conditional store buffer. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 160–169, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[86] Lambert Schaelicke and Alan L. Davis. Design trade-offs for user-level i/o architectures. *IEEE Trans. Comput.*, 55:962–973, August 2006.

[87] Steven W. Schlosser, John Linwood Griffin, David F. Nagle, and Gregory R. Ganger. Designing Computer Systems with MEMS-based Storage. *SIGOPS Oper. Syst. Rev.*, 34(5):1–12, 2000.

[88] Frank B. Schmuck and Roger L. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *USENIX Conference on File and Storage Technologies*, pages 231–244, 2002.

[89] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the 1990 Winter Usenix*, pages 313–324, 1990.

[90] Eric Seppanen, Matthew T. OKeefe, and David J. Lilja. High performance solid state storage under linux. In *Proceedings of the 30th IEEE Symposium on Mass Storage Systems*, 2010.

[91] Cxfs. http://www.sgi.com/products/storage/software/cxfs.html.

[92] Prashant J. Shenoy and Harrick M. Vin. Cello: a Disk Scheduling Framework for Next Generation Operating Systems. *SIGMETRICS Perform. Eval. Rev.*, 26(1):44–55, 1998.

[93] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Symposium on Mass Storage Systems*, 2010.

[94] Steven R. Soltis, Grant M. Erickson, Kenneth W. Preslan, Matthew T. O'Keefe, and Thomas M. Ruwart. The Global File System: A File System for Shared Disk Storage. *IEEE Transactions on Parallel and Distributed Systems*, 1997.

[95] Jeremy Sugerman, PUanesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[96] http://sysbench.sourceforge.net/index.html.

[97] Toby J. Teorey and Tad B. Pinkerton. A Comparative Analysis of Disk Scheduling Policies. *Commun. ACM*, 15(3):177–184, 1972.

[98] Alexander Thomasian and Chang Liu. Disk Scheduling Policies with Lookahead. *SIGMETRICS Perform. Eval. Rev.*, 30(2):31–40, 2002.

[99] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, Palo Alto, 2008.

[100] Mustafa Uysal, Arif Merchant, and Guillermo A. Alvarez. Using MEMS-Based Storage in Disk Arrays. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 89–101, Berkeley, CA, USA, 2003. USENIX Association.

[101] Violin memory 6000 series flash memory arrays. http://www.violin-memory.com/products/6000-flash-memory-array/.

[102] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing. *SIGOPS Oper. Syst. Rev.*, 29:40–53, December 1995.

[103] http://www.symantec.com/cluster-file-system.

[104] WD VelociRaptor: SATA Hard Drives. http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701284.pdf.

[105] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[106] Western Digital. WD Caviar Block Desktop Hard Drives, 2010. http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701276.pdf.

[107] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 306–317, Washington, DC, USA, 2007. IEEE Computer Society.

[108] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling Algorithms for Modern Disk Drives. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 241–251, New York, NY, USA, 1994. ACM.

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

[110] Lei Xia, Jack Lange, Peter Dinda, and Chang Bae. Investigating virtual passthrough i/o on commodity devices. *SIGOPS Oper. Syst. Rev.*, 43:83–94, July 2009.

[111] Inc. Xilinx. Virtex-5 Endpoint Block Plus Wrapper for PCI Express (PCIe). http://www.xilinx.com/products/intellectual-property/V5_PCI_Express_Block_Plus.htm.

[112] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *in proceedings of the 10th USENIX Conference on File and Storage Technologies*, February 2012.

[113] Weikuan Yu, Shuang Liang, and Dhabaleswar K. Panda. High performance support of parallel virtual file system (pvfs2) over quadrics. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 323–331, New York, NY, USA, 2005. ACM.

[114] Yuanyuan Zhou, Angelos Bilas, Suresh Jagannathan, Cezary Dubnicki, James F. Philbin, and Kai Li. Experiences with VI communication for database storage. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 257–268, Washington, DC, USA, 2002. IEEE Computer Society.